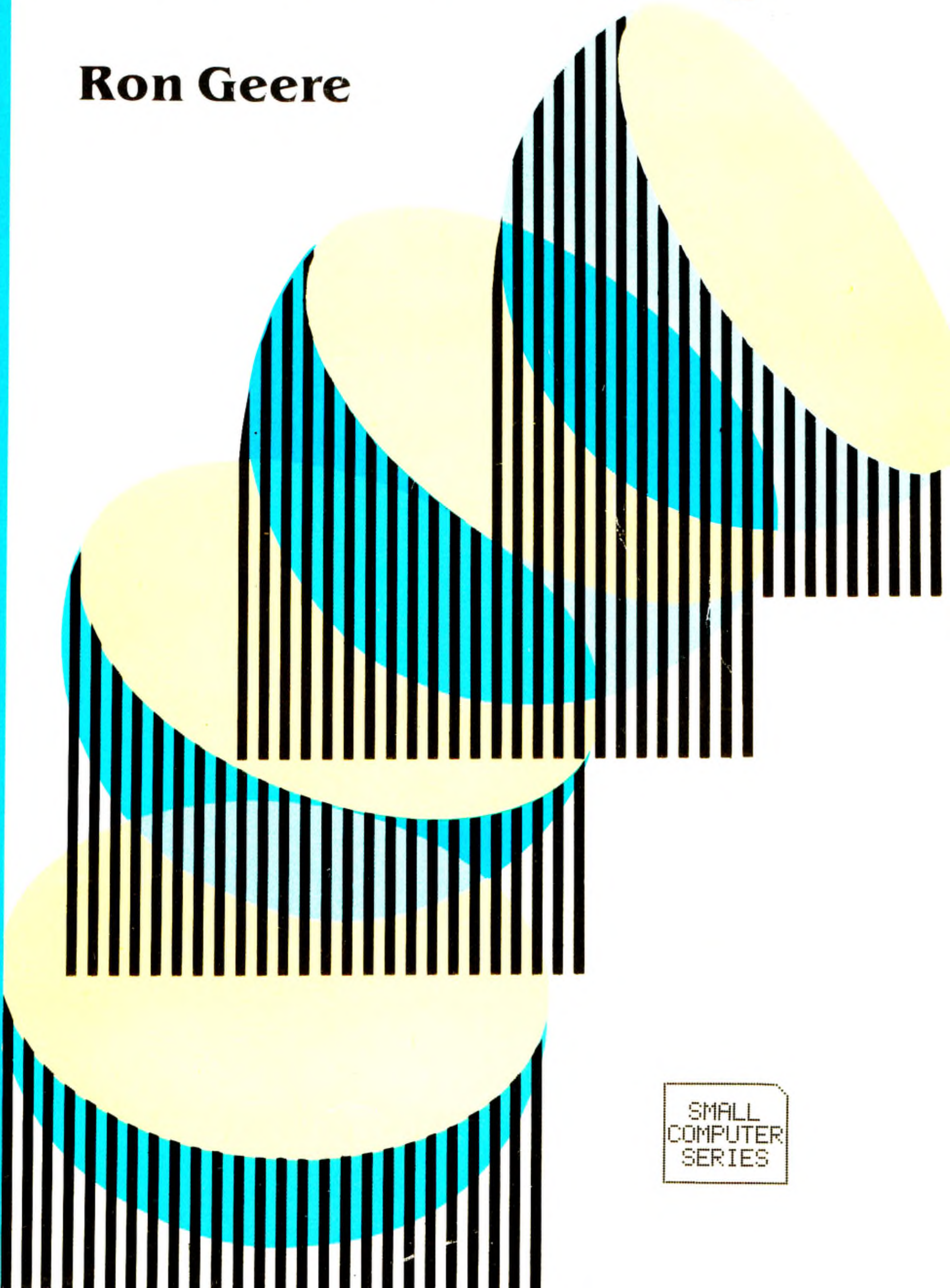
Forth: The NEXT Step

Ron Geere



SMALL COMPUTER SERIES

Forth:
The NEXT Step

SMALL COMPUTER SERIES

Consulting Editors	**Jan Wilmink** ORMAS bv, The Netherlands
	Max Bramer Thames Polytechnic

Forth: The NEXT Step

Ron Geere

ADDISON-WESLEY PUBLISHING COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
Don Mills, Ontario · Amsterdam · Sydney · Singapore · Tokyo · Madrid
Bogota · Santiago · San Juan

© 1986 Addison-Wesley Publishers Limited
© 1986 Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of the publisher.

The programs presented in this book have been included for their instructional value. They had been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs.

Cover design by Ken Arlotte.
Typeset by Leaper & Gard Ltd.
Printed in Great Britain by The Hollen Street Press.

Library of Congress Cataloging in Publication Data

Geere, Ron.
FORTH, the next step.

(Small computer series)
Bibliography: p.
Includes index.
1. FORTH (Computer program language)
I. Title II. Series.
QA76.73.F24G44 1986 005.13'3 85-30805
ISBN 0-201-18050-2 (pbk.)

British Library Cataloguing in Publication Data

Geere, Ron
Forth: the next step.—(Small computer series)
1. FORTH (Computer program language)
I. Title II. Series
005.13'3 QA76.73.F24

ISBN 0-201-18050-2

ABCDE 89876

Preface

Most computer languages appear to have a field of use for which they are most appropriate. Forth is no exception and has become the standard language of the International Astronomical Union and is used for the control of radio telescopes. Forth naturally finds its way into other control applications such as process control, machine tool control and robotics. In addition it has been used in diverse applications such as video games, spreadsheets and the space shuttle aft flight deck simulator. A major advantage of Forth is that it can be tailored to the application, rather than the converse. In essence, Forth provides us with the tools to do the job.

Like the English language, Forth is extensible. What is meant by that? Suppose by way of example we invent a completely new item, we will therefore need to call it something, say a 'whatsit'. We can now define what we mean by a 'whatsit' in terms of other words that we already know. To find the meaning of these other existing words we can look them up in a dictionary, although this presupposes that we already know a minimal number of words. In principle this is what we do in Forth.

There are many books on Forth which go to great pains to explain the minimum required word set and give a few examples of their use. However when it comes to progressing beyond this stage it depends very much on one's own ability. At this point if the programmer does not take to Forth naturally, as can happen when weaned on a more conventional procedural language, it is tempting to say, 'Hmm, very interesting' and then go back to one's familiar language. After all, compared with a level-2 BASIC, Forth lacks the comfortable line numbers, floating point arithmetic, trigonometric functions, strings and arrays. But Forth is extensible. If the user finds it necessary the required features may be added in a form most suited to the application.

In this book some common extensions are presented, together with some programs that have been found either useful or entertaining. It is hoped that in so doing the methods will stimulate readers to see solutions to their own programming problems. The extensions are commonplace and may be standard in some Forths but lacking in others. Inevitably when seeking a missing definition in a hurry, one never seems able to find it readily when it is most needed. Hopefully the collection in this book will fulfil that need.

One of Forth's many virtues is that it is far more portable than languages such as BASIC, the latter having over 100 differing dialects. This portability is achieved by first defining a virtual Forth computer for a given machine. This simulated computer now looks the same for every computer so that the same Forth language will run on it. There are, however, variants in the language and some word definitions can be in the native code of the machine in use and probably will not transfer directly to a different processor type. These Forth dialects are usually not too different but where a difference is known to exist, this has been noted in the text. Additionally most differences are between FigForth and the '79-standard and are listed in Chapter 1. At the time of writing these two standards, although superseded, are still dominant in number. Variants considered are FigForth, '79-standard and '83-standard. Commercial variants such as MMS-Forth and PolyForth are not covered in this book because their comprehensive vendor support renders it unnecessary. Where `CODE` definitions are used, 6502 code is used with high-level Forth equivalents also included where applicable. Although machine-specific, this processor was chosen because it is used by Apple, Atari, Commodore and the BBC computers.

Finally the author would like to express appreciation to all who have put so much effort into publicizing Forth and putting it to work on an ever increasing range of tasks.

Ron Geere *Farnborough*

Contents

1 Getting Organized

Mass storage is a key element in Forth and is sometimes referred to as 'virtual memory'. This is a concept whereby the computer is led to believe that it has more memory than it really has. It works by storing the program or data on a mass storage device, such as a floppy disk drive, and transferring only part of the total storage available into the computer's memory space at any one time. Historically this came about so that Forth could be used on small microprocessor systems with limited memory capacity. Recent 16-bit machines have ample memory for Forth and versions exist which have dispensed with the virtual memory arrangement. At the other extreme some low cost systems dispense with disk storage in favour of cassette storage. Such systems are perhaps adequate for learning and experimentation, but cannot be considered for serious program development.

A 'bare bones' disk-based Forth system will require some initial preparation of the disk itself. Starting off with a totally unused diskette, the first job will be to format it with tracks and sectors. This is not specific to Forth, but a characteristic of 'soft-sectored' disk usage. To do this from Forth, you will need to refer to your user handbook. If your Forth is a 'home brew' system, you will have to home brew your own method of doing this also!

The next step is to put the following Fig and '79-standard error messages on screens 4 and 5 using the `EDITOR`. The text of line number 15 of screen 4 is printed at the foot of each page by `TRIAD` and is usually modified to contain a copyright message or some personal identification.

```
SCR #4
 0 (  ERROR MESSAGES  )
 1 EMPTY STACK
 2 DICTIONARY FULL
 3 HAS INCORRECT ADDRESS MODE
 4 ISN'T UNIQUE
 5
 6 DISK RANGE ?
 7 FULL STACK
 8 DISK ERROR !
 9
10
11
```

to be inserted into the sequence. One way round this problem is to keep the chained blocks to perhaps two or three related blocks and to call the head of each chain from a 'load block'.

The purpose of a load block is to call up and load all the blocks required for a specific application. It is convenient if the block/screen number used is numbered relative to the load block. To do this we define RLOAD (or perhaps +LOAD) which will load a block calculated from the number of the load block plus a signed offset on the stack. The definition is also arranged to print the header (line 0) of the block at the head of each chain, spaced for an 80-column VDU.

```
: RLOAD     ( N --- /Load scr+N                                    *)
            BLK @ +  CR 8 SPACES  0 OVER .LINE
                     CR 2 SPACES  LOAD  ;
```

A hypothetical load block might have some of the following information.

```
 3 RLOAD  ( editor extensions )
 8 RLOAD  ( Forth extensions )
12 RLOAD  ( 2* 2/ etc. )
15 RLOAD  ( double word set )
29 RLOAD  ( trig functions )
45 RLOAD  ( application constants )
57 RLOAD  ( system input routines )
43 RLOAD  ( data validation )
33 RLOAD  ( output formatting )
```

You will see that each subsection is commented. It also makes life easier if the load block is at an easily remembered number – 10 LOAD is easier to remember than, say 19 LOAD. One could even go a step further and define a constant for each of several load screens using the application name for each as a label. As previously mentioned, the first screen line is displayed as each RLOAD is executed, so that when the application loads it might look like this:

```
        ( editor extensions - 1 of 4  search  replace      RDG-820211 )
  size ?isn't unique
        ( extensions - 1 of 2   ascii  $xx  ctrl-x         RDG-830115 )

        ( code for 2s - 1 of 3    2*  2/  2**              RDG-820224 )

        ( double word set - 1 of 3                         RDG-830507 )

        ( trig functions - 1 of 4  sin cos etc.            RDG-830515 )

        ( constants - 1 of 1                               RDG-830521 )

        ( system input - 1 of 2                            RDG-830522 )

        ( data validation - 1 of 2                         RDG-830630 )

        ( formatting - 1 of 3                              RDG-830702 )
```

If the hard copy printer supports 132-column output it is useful to be able to print two screens side by side. This is especially so if shadow blocks are being used since the description can be adjacent to the definition. Blocks can be listed in pairs by using PLIST defined in Chapter 10.

A number of useful Forth words have been published which enhance the readability of a program: for example, how much more readable it is if one defines the words:

```
: TRUE  1  ;  : FALSE  0  ;
```

or perhaps replaces 3 EMIT with CTRL-C EMIT or 42 EMIT with ASCII * EMIT – such words are a prerequisite in any sizeable application.

```
: ASCII  ( --- N /Leave ASCII value of next in-line char.        *)
         BL WORD  HERE 1+  C@  [COMPILE] LITERAL  ; IMMEDIATE
```

ASCII leaves on the stack the ASCII number equivalent of the following character [1]. When compiling, the next input-stream WORD delimited by BLank is placed at HERE and the first character fetched to the stack for use by LITERAL whose compilation is forced by [COMPILE]. A similar, shorter, but less readable word is &X where X = the character [2],[3]. In this case the parameter following is not passed via the stack but by way of the header with the WIDTH reduced to one. $XX simplifies hexadecimal notation since the base can remain unchanged and the sequence 8 16 32 64 may be better expressed as $08 $10 $20 $40 in some instances. Likewise addresses are usually best expressed in hexadecimal as $E000 for example, without perforating the text flow alternately with the words HEX and DECIMAL. HERE should be included only for FigForth.

```
( &X  $XX  $XXXX  CTRL-X                                    )
            32 CONSTANT &       1 WIDTH !
: &X        HERE 2+ C@ [COMPILE] LITERAL ;          IMMEDIATE

: $XX       BASE @ HEX 0 0 HERE 1+ (NUMBER) SWAP DROP ROT BASE !
            C@ BL - 0 ?ERROR [COMPILE] LITERAL ;    IMMEDIATE

: $XXXX
            [COMPILE] $XX ;                         IMMEDIATE

            5 WIDTH !
: CTRL-X
            HERE 6 + C@ 64 - [COMPILE] LITERAL ;    IMMEDIATE
  31 WIDTH !


```

In later standards (NUMBER) should be replaced by CONVERT. Additionally FigForth users may want to add definitions which are required in later standards but which are lacking in FigForth, such as U. M+ followed by some of the extensions defined in later chapters.

1.3 DIFFERENCES BETWEEN FIGFORTH AND '79-STANDARD FORTH

Forth originated from the ideas of one man but as the language has matured several standards have evolved. Unfortunately it is not practicable to give cross-conversion between any two of the major dialects. At the time of writing, FigForth and the '79-standard are numerically the most popular and, therefore, if you are using Forth based on the early Forth Interest Group (Fig) standard you may have to make changes to a program written to the '79-standard. In general a simple conversion process is all that is involved, but some words such as DOES> behave differently and are perhaps best handled by an amendment to the program if necessary.

```
SCR #133
( '79 STANDARD CONVERSION TO FIGFORTH                          )
CODE J
                  XSAVE STX, TSX,
                  R 4 + LDA, PHA, R 5 + LDA,
                  XSAVE LDX, PUSH JMP,

CODE R@
                  FORTH  ' R CFA @  ' R@ CFA !

CODE EXIT
                  ' ;S CFA @ ' EXIT CFA !

-->



```

```
SCR #134
( '79 STANDARD CONVERSION TO FIGFORTH                          )

: MOVE  ( AD1 AD2 N --- )
                  2 * CMOVE  ;

: VARIABLE
                  0 VARIABLE  ;

: CREATE
                  VARIABLE -2 ALLOT  ;
-->





```

```
SCR #135
( '79 STANDARD --> FIGFORTH )
: U.              0 D.  ;
: 0>              0 >   ;
: 1-              1 -   ;
: 2-              2 -   ;
: >IN             IN    ;
: ?DUP            -DUP  ;
: CONVERT         (NUMBER)  ;
: DNEGATE         DMINUS  ;
: NEGATE          MINUS   ;
: NOT             0=   ;
: SAVE-BUFFERS    FLUSH   ;
: U/MOD           U/   ;
: SIGN            0< IF  2D HOLD  THEN  ;
: WORD            WORD HERE  ;
: 79-STANDARD ;
```

The '83-standard may present further problems with a different behaviour associated with important words such as CREATE, LEAVE, EXPECT, WORD, FIND, PICK and ROLL. As a result conversion to or from Forth-'83 is far more complicated. For further information you should refer to the appropriate Forth standards [4] or the articles by Berkey [5],[6]. Reference [7] explains the differences in some detail.

'All changes to a computer programming language degrade its quality'
– Professor A. Sale

1.4 AVOIDING THE ASSEMBLER

Sometimes it is necessary to produce a definition in the native code of the processor. To do this on a small system for perhaps only one definition is somewhat inefficient in memory usage, since the entire assembler must be loaded for just that one item. Alternatively, on a small system, there may not even be an assembler available!

So how can we avoid using the memory space required by the assembler, yet include CODE definitions in our vocabulary? Suppose we want to use the word RVS to invert the video of a memory-mapped CRT screen. Using a typical assembler the word could be defined thus:

```
CODE RVS   ( --- /Invert 2K of video RAM starting at $8000        *)
           XSAVE STX,  $08 STY,  88 # LDA,  09 STA,
           BEGIN,  09 DEC,
                   BEGIN,  08 )Y LDA,  80 # EOR,  08 )Y STA,
                           DEY,  0=
                   UNTIL,  09 LDX,  80 # CPX,  0=
           UNTIL,  XSAVE LDX,
           NEXT JMP,   END-CODE
```

This would compile into the dictionary a header containing the name RVS and a parameter field containing bytes corresponding to the assembled code. The same effect could be achieved by the following method:

```
( UTILITY - RVS     INVERTS SCREEN VIDEO                  RDG-830729 )
HEX
CREATE RVS ( --- /Invert $8000 to $7FFF                           *)
            0586 , 0884 , 88A9 , 0985 , 09C6 , 08B1 , 8049 , 0891 ,
            88  C, F7D0 , 09A6 , 80E0 , EFD0 , 05A6 , 4C  C, 0642 ,
            SMUDGE DECIMAL
```

The word CREATE creates a dictionary header containing the name RVS. The succeeding numbers are then compiled into the dictionary and examination will reveal that they correspond to the same numbers that would be generated by the assembler. Obviously some manual conversion must be undertaken, especially with branch offsets, for this is what assemblers are intended to eliminate. One needs, therefore, a table of op-codes for the processor concerned (in this example the 6502) and also the address of the word NEXT.

If you are interested in using RVS you are probably aware that it was written for the Commodore 8032 which has 2K of video-mapped RAM from $8000 to $87FF and the routine uses an address pointer at $08/09. A word of caution: this definition is even more unreadable than the version using the assembler. Do document how it works on the disk or you will waste a lot of time someday trying to fathom it all out. Even with a disassembler to convert back to assembly language, some additional notes will not go amiss.

The definitions :CODE and ;NEXT make the creation of these hand assembled definitions neater and easier to read at a later date.

```
( WORDS IN LIEU OF ASSEMBLER                        RDG-841014 )

HEX    0642 CONSTANT NEXT-LINK    ( the address of NEXT )

: :CODE ( --- N /CREATE HEADER WITH CFA
                        POINTING TO BODY OF WORD                  *)
        BASE @  HEX  CREATE  ;

: ;NEXT ( N --- /Terminate body of word with jump to NEXT      *)
        4C C,  NEXT-LINK ,  SMUDGE     BASE !  ;
DECIMAL
```

If you have the source listing of an assembler for your system, you may be able to find the address of NEXT from that. Alternatively, other related words may be required and to find them on a typical 6502-based system you can try these assumptions and leave the address on the stack.

```
HEX ' LIT  11 + CONSTANT PUSH          (push lo in A, hi in P)
    ' LIT  13 + CONSTANT PUT           (drop top & PUSH)
    ' LIT  18 + CONSTANT NEXT
    ' EXECUTE NFA 11 - CONSTANT SETUP
    ' (DO) 0C + CONSTANT POPTWO        (remove 2 items)
    ' (DO) 0E + CONSTANT POP           (remove 1 item)
```

They are correct for FigForth, and the addresses are assumed to be fixed offsets from known landmarks in the dictionary kernel. However, should the source code differ in some respect from the FigForth source the assumption may not be valid. In addition, the FigForth word tick (') which returns the parameter field address of the following word may need to be replaced by `FIND`. Since also in FigForth, `FIND` returns the code field address the figures will also need adjustment.

It is a wise precaution before testing that the code definition operates as intended to first check the compiled code using `DUMP`. The most likely error with 6502 code is to have hi/lo byte pairs transposed or perhaps relative branch/offset values reversed.

`CREATE` is a '79-standard word and may not be in a FigForth implementation; as an alternative to re-defining it (see `SCR 134` in Section 1.3) `CREATE` can be replaced by:

```
0 VARIABLE <name>  -2 ALLOT
```

Project

Explore your dictionary to find the addresses for `NEXT` etc. Expand the pseudo-assembler to include `;PUSH ;PUT ;POP` – can a defining word `ENDING` (created with `CREATE...DOES>`) be used for this family of words? e.g. `NEXT-LINK ENDING ;NEXT`

REFERENCES

[1] Weisling, R. *Forth Dimensions*, **III**, No. 3, p72
[2] $XX etc., *Forthwrite 2, No. 5*, U.K. Forth Interest Group. Originally from L.A. Fig Users.
[3] Huang, T. 'In-word Parameter Passing' *Forth Dimensions*, **V**, No. 3, p19
[4] *The Forth-'83 Standard.* Forth Interest Group, P.O. Box 1105, San Carlos, CA94070, U.S.A.
[5] Berkey, R. 'Upgrading Forth-'79 Programs' *Forth Dimensions*, **VI**, No. 3, p26
[6] Berkey, R. 'Forth-'83 Program to Run Forth-'79 Code' *Forth Dimensions*, **VI**, No. 4, p28
[7] McCabe, K.C. August 1984. 'Forth-'83: Evolution Continues' *BYTE Magazine*, p137

2 Some Forth Extensions

There are many definitions that are commonplace but do not form part of the minimum required word set. In this section an assortment is presented that you may find useful.

2.1 COUNTER-ROTATE (-ROT OR <ROT)

This is often required after operating on the third stack item. ROT brings it to the top and ROT ROT reinstates it. One could define -ROT as ROT ROT, but this version is more efficient.

```
: -ROT      ( N1 N2 N3 --- N3 N1 N2 /Counter-rotate top three     *)
            SWAP  >R  SWAP  R>  ;
```

2.2 TUCK

A combination of words that occurs frequently gives rise to the definition TUCK. This puts a copy of the top stack item under the top two. It could be considered as the converse of OVER, so perhaps it ought to be called UNDER?

```
: TUCK      ( N1 N2 --- N2 N1 N2 /Copy TOS under top two            *)
            SWAP  OVER  ;
```

2.3 LOOKING AT THE STACKS

DEPTH appears in some systems; if not, the following definition will return the number of items on the stack that were there before DEPTH was executed. Note that this definition is system dependent. The numbers 136 and 134 are related to the base address of the stack of a 6502 system. The 136 could be replaced by S0, the contents of user variable SP0. Some implementations have S0 and/or SP0 missing, and even if it is available, the detail of its operation may vary across different Forth standards. Entering

```
SP!  DEPTH  .
```

should print 0; otherwise adjust these values until it does.

```
: DEPTH      ( --- N /Leaves number of items on the stack            *)
             SP@  136  SWAP -  2 /  ;
```

There are some Forth systems that use 'S instead of SP@ to return the address to the top of the stack as it was before the address was added.

To print out the stack contents non-destructively, the definition .S has proved itself useful. Again, it is system dependent:

```
: .S         ( --- /Prints the stack contents                        *)
             CR  DEPTH IF   SP@ 2 - 134 DO  I @ . -2 +LOOP
                      ELSE ." Empty"  THEN  ;
```

The Forth-'79 and subsequent standards prohibit access to Forth's innards so that RP@ which returns the current value of the return stack pointer is no longer part of the required word set. However, it is a useful word when developing utilities which hook into Forth's inner workings:

```
CODE RP@     XSAVE STX,  TSX,  TXA,  XSAVE LDX,  PHA,  01 # LDA,
             PUSH JMP,   END-CODE
```

which without the assembler is equivalent to:

```
HEX
CREATE RP@   ( --- N /Returns address in return stack register      *)
             0586 , BA C, 8A C, 05A6 , 48 C, 01A9 ,
             4C C, 063B , ( PUSH )  SMUDGE  DECIMAL
```

PICK is not included in the required word set of FigForth, although extensive use of this word may imply that perhaps there are too many items on the stack for good Forth style.

```
: PICK       ( N1 --- N2 /Pick the N1th stack item & copy to top     *)
             DUP +  SP@ +  @  ;
```

Be careful of programs written for the '83-standard where the value of N1 is zero-based. In Fig and '79-standards the top stack item is number one. An example of the use of PICK is in 3DUP which duplicates the top three stack items.

```
: 3DUP       ( N1 N2 N3 --- N1 N2 N3 N1 N2 N3 /DUPlicate top 3       *)
             3 PICK  3 PICK  3 PICK  ;
```

or alternatively:

```
DUP  2OVER  ROT  ;
```

A further definition of 3DUP appears in DSQRT in Chapter 6. One should of course be less specific and define a more general purpose version called NDUP which replicates the top *N* stack items:

```
: NDUP       ( N --- /Replicate top N stack items                    *)
             1+ DUP 1  DO  DUP PICK SWAP  LOOP  DROP  ;
```

2.4 LOOP VARIABLES

With the more recent microprocessors it is possible that DO...LOOP parameters have their own independent stack and the following system dependent definitions will need to be rewritten. They assume that the loop stack and the return stack are the same. This is why the words I and R (or R@) are not necessarily identical.

The word I' is commonly used to retrieve the loop limit from within the loop.

```
: I'        ( --- N )
            FORTH  R> R>  R  -ROT  >R >R  ;
```

The top item on the return stack is the address of the next definition to be executed (usually, but not always) and the second item is the loop variable. R or R@ fetches a copy of the loop limit. -ROT puts this limit under the other two values which are then restored to the return stack. See Figure 2.1.

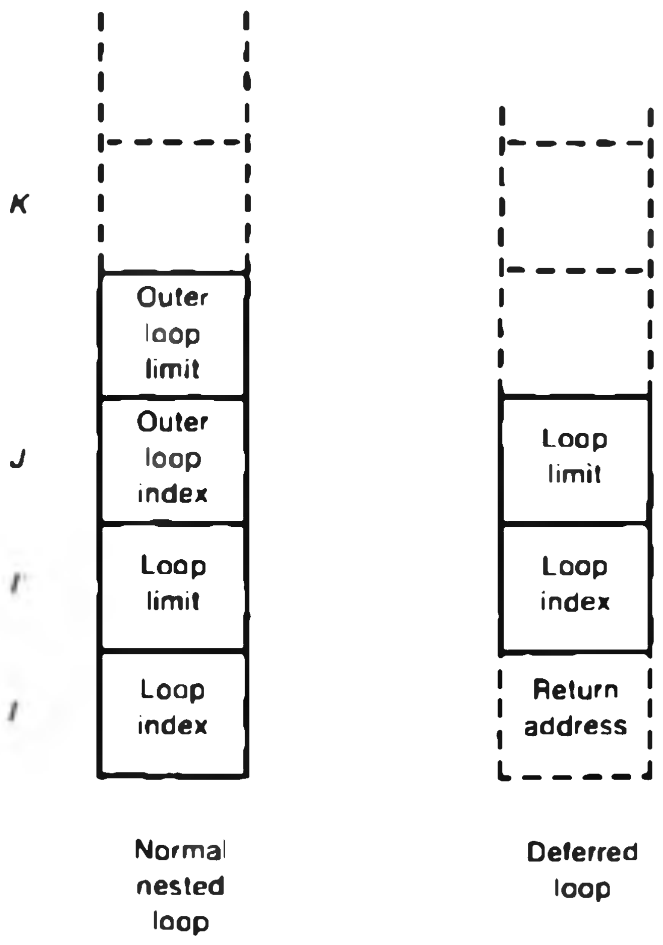


Figure 2.1 Loop parameters on the return stack.

An alternative use for I' is to obtain the loop variable from another definition used within the loop. For example:

```
: (TEST)   CR  I' 3 .R  ;
:  TEST    5 0 DO  (TEST)  LOOP  ;
```

J returns the value of the outer loop variable of a nested DO...LOOP:

```
HEX
CREATE J    0586 , 8A C, 8D C, 0105 , 48 C, 8D C, 0106 ,
            05A6 , 4C C, 063B , ( PUSH )    SMUDGE  DECIMAL
```

which in assembler is:

```
XSAVE STX,   TSX,   RP) 4 + LDA,   PHA,
RP) 5 + LDA,   XSAVE STX,   PUSH JMP,
```

This could be coded in high-level thus:

```
: J         R>  R> R>  R   SWAP >R  SWAP >R  SWAP >R  ;  ( --- N )
```

The high-level version respectively removes from the return stack the cfa arising from J itself, inner loop variable I, inner loop end limit and then copies the outer loop index. These are then in turn each SWAPped and returned so that the stack may be correctly restored. Take care not to confuse R> and >R – the result would be catastrophic!

BOUNDS will convert the count *N* at the given address to start and end values ready for a DO . . . LOOP.

```
: BOUNDS    ( addr N --- addr+N addr /Make start & end values     *)
            OVER  +  SWAP  ;
```

2.5 RECURSIVE ROUTINES

Recursion requires a routine to call itself. In some computer languages this cannot be done, but Forth is not so constrained. Recursion makes demands on the return stack and in extreme cases this could be a limitation.

In Forth, the word currently being compiled is not 'seen' to be in the vocabulary until its definition is complete. Many of you will know that the purpose of the 'smudge bit' is to identify this situation. The problem is to compile the cfa of a word which has yet to be SMUDGEd. Fortunately the solution is simple because LATEST returns the name field address (nfa) of the latest definition without reference to the smudge bit. To convert this address to the compilation or code field address (cfa) we must first convert the nfa to a parameter field address (pfa) and this in turn to the cfa which is subsequently compiled. Finally the definition is made IMMEDIATE so that it acts during the compilation process.

```
: MYSELF    ( --- /Calls current definition from within itself    *)
            LATEST  PFA  CFA ,  ; IMMEDIATE
```

Sometimes this word is named RECURSE. As an example of its use, here is a recursive definition of ROLL [1].

```
: ROLL      ( Rotate the top N stack items                          *)
            DUP 2 < IF    DROP
                    ELSE  SWAP >R  1 - MYSELF R>  SWAP
                    THEN  ;
```

The definition of ROLL sets out to eliminate the trivial case where N = 1 or less. It then moves in turn each former stack item to the return stack and decrements N by one each time. The process is repeated by calling itself (remember MYSELF compiles the cfa of ROLL) until the conditional becomes true (N < 2) and MYSELF is no longer called. The expression following MYSELF restores each stack item, but SWAP the *N*th item to the top until the return stack is restored.

The same comments apply to ROLL as to PICK regarding the stack parameter, but then an '83-standard implementation should already have both PICK and ROLL.

2.6 MEMORY USAGE

To determine the amount of available memory use:

```
: FREE     ( --- /Return bytes free in dictionary                *)
           FIRST  HERE  - . ." BYTES"  CR  ;
```

since this is simply the difference between the first disk buffer and the current top of the dictionary. However it is dependent on the layout of the Forth model for the system in use and takes no account of space occupied by PAD for example.

2.7 TESTING PARAMETERS

While Forth's do-it-yourself nature can be one of its virtues, it can lead to a proliferation of variations on the same theme. The definition of WITHIN in various guises has appeared from time to time [2] and is used to test that a parameter lies WITHIN a specified range. The input stack parameter sequence may be different, and the limits may be equal or exclusive to the value of N. Of course the polarity of the output flag could also differ! The example shown was chosen for its brevity. It is not the most efficient because most of the coding is hidden within the words MIN and MAX.

```
: WITHIN   ( lo hi N --- f /True flag if N is = or within lo-hi  *)
           DUP >R  MIN MAX  R> = ;
```

My personal preference is for TWIXT which leaves a true flag if lo ≤ N ≤ hi.

```
: TWIXT    ( N lo hi --- f /flag is true if N is twixt lo & hi   *)
           3 PICK  <  0=  -ROT  <  0=  AND  ;
```

On occasions it is necessary to constrain a value to within certain limits. LIMITS is neat and simple, most of its work being hidden inside MIN and MAX. An alternative name would be CONFINE.

```
: LIMITS   ( N1 lo hi --- N2 /N2 is N1 limited to lo & hi         *)
           ROT  MIN  MAX  ;
```

Similar to LIMITS is <MOD> except that a wrap around effect occurs as with MOD, although the behaviour is different. The table illustrates the action of <MOD> for N2 = 5:

```
Input:   -1  0    1  2  3  4  5    6  7  8
Output:   4  5::1 2  3  4  5::1 2  3  etc.
```

A likely use for such action is to increment or decrement the index to an array or table with wrap around. Index position zero in each of several such tables contains the modulus N2. <MOD> then constrains N1 to the index limits of that table.

```
: <MOD>    ( N1 N2 --- N3 /limits N1 to range 1 - N2 as value N3 *)
           SWAP  1 - OVER  MOD  DUP 0<
           IF  +  ELSE  SWAP DROP  THEN  1+  ;
```

2.8 ROUNDING

Errors often occur as a result of integer division where the effect is to truncate the fractional part. The error may be reduced by scaling the numerator by 10 if possible and applying ROUND to the result. The effect is to round up 175 to 180 and round down 174 to 170. The scaling factor may be removed later. Notice that with negative values −175 goes to −180 and −174 rounds to −170.

```
: ROUND    ( N1 --- N2 /Leaves N2 as N1 rounded to nearest 10     *)
           10  /MOD  SWAP  5 / +  10 *  ;
```

2.9 SIGNUM

This function returns the sign of a number, or zero if the number is zero.

```
: SGN      ( N1 --- N2 /Return sign if non-zero, else zero        *)
           DUP  IF  1 SWAP  +-   THEN  ;
```

If zero is on the stack, the condition is false and the zero remains, otherwise +- applies the sign of the stack item to 1 which is left instead.

2.10 NOT WHAT IT SEEMS

Boolean values are either false or true, i.e. represented by 0 for false and either +1 or −1 for true. The logical operators AND OR and XOR act on all corresponding bits in a 16-bit value (or 'cell' in Forth terminology). In so doing they leave a 16-bit value as opposed to a two-valued Boolean flag.

In some Forths, NOT is defined identically to 0= but in practice this is not always so. NOT is sometimes defined to perform a one's complement operation in which case NOT applied to a positive number which is a Boolean true leaves a negative number which is also a Boolean true – NOT what was intended. It is safer to use 0= which is what it says, true if non-zero, else false. The complement of this is 0<> and is defined simply:

```
: 0<>        0= 0=  ;
```

Suppose we have the situation where we want a true flag to result from a logical AND of two given non-zero numbers. It is tempting to assume that each number is a Boolean true and therefore the logical AND of the two numbers will suffice. A moment's thought will show that a statement such as: 4 2 AND is false. The bit patterns for 4 and 2 do not have any 1s in common. What is required is 0<> after each number to convert the non-zero number to a Boolean true flag. The logical AND may then be correctly performed on the flags.

Note also that while a Boolean false is zero, a true flag may be 1 in early standards and −1 (all bits set) in later standards. Finally, you should never mix logical 'values' with arithmetic values in computations. Such attempts would probably not work correctly on all Forth systems and would be difficult to understand at a later date.

2.11 HAVING DONE

Use of the word EXIT is not considered to be good programming practice since this unstructured word will leave a definition prematurely. However, it has its uses and can sometimes make a neater job of a complex multiple structured word. It may *not* be used in a DO ... LOOP where the use of LEAVE should be considered instead.

```
: EXIT     ( --- /Exit immediately from the current definition   *)
         . R>  DROP  ;
```

By simply dropping the address on the return stack, after EXIT the interpreter goes to the next word after the semicolon of the word using EXIT. Use it, if you must, with caution. Make sure that the return stack does contain what is expected on top, not something arising from >R or a loop. Note also that in '79-standard and later standards EXIT is sometimes used outside a definition instead of \S or FigForth's ;S to signify when to stop compiling a screen.

2.12 FAMILY OF WORDS TO MANIPULATE BYTES IN A CELL

These words are useful for data packing, sorting, and graphics byte manipulations. The first here – CSWAP – interchanges the high and low bytes of the top stack value.

```
CODE CSWAP      TOP LDA,   TOP 1+ LDY,   TOP 1+ STA,   TOP STY,
                NEXT JMP,   END-CODE
```

Defined without the assembler, this becomes:

```
HEX
CREATE CSWAP    ( N1 --- N2 /Swap hi & lo bytes                    *)
                00B5 , 01B4 , 0195 , 0094 , 4C C, 0642 , SMUDGE
DECIMAL
```

The word CSPLIT takes the high and low bytes of the top stack item and leaves them on the stack as two separate items:

```
: CSPLIT        ( N --- HI LO /SPLIT HI & LO AS TWO ITEMS        *)
                0  256 U/  SWAP   ;
```

Firstly it is necessary to make it an unsigned double precision number so that the high order bit is not regarded as the sign. U/ leaves a remainder and quotient corresponding to the two bytes concerned. Alternatively CSPLIT may be defined in code:

```
CODE CSPLIT     DEX,  DEX,
                 TOP  2+ LDA,   TOP    STA,
                 TOP 3 + LDA,   TOP 2+ STA,
                 TOP 3 + STY,   TOP 1+ STY,   NEXT JMP,  END-CODE
```

Naturally if there is a need to split a word, it follows that there will be a requirement sometime to join two separate values each less than 256 and representing the required high and low byte values of the joined word.

```
: CJOIN         ( HI LO --- N /JOIN TWO WORDS AS ONE             *)
                SWAP  256 *  OR  ;
```

In most instances the sequence of split bytes is quite arbitrary, so that if lo/hi is preferred, both occurrences of SWAP can be omitted. However, if the CODE definition is being employed, some rewriting will be necessary.

When values are stored in variables or arrays, the contents of two addresses may need to be swapped as is done in some sort routines. VSWAP will exchange the two values.

```
: VSWAP         ( addr1 addr2 --- /Swaps the address contents    *)
                DUP >R  @ SWAP DUP   @ R>  !  !   ;
```

The operation of VSWAP is straightforward.

2.13 A COMPLEMENT OF TWOS

Frequently definitions are required that involve the figure 2. This section covers an assortment of such definitions.

```
: 2*        ( N --- 2*N /Doubles top of stack value              *)
            DUP  +  ;
```

The above definition simply doubles the top stack value, be it positive or negative. However there are occasions when an 'arithmetic shift left' is required and the high level definition becomes more complicated and too slow. The following CODE definitions are presented as an alternative:

```
CODE 2*     ( U --- 2*U /does arithmetic shift left              *)
            TOP ASL,  TOP 1+ ROL,  NEXT JMP,  END-CODE
```

Or without requiring the assembler:

```
HEX
CREATE 2*  ( U --- 2*U /Return unsigned times 2 [ASL]              *)
           0016 , 0136 , 4C C, 0642 ,  SMUDGE   DECIMAL
           ( TOP ASL,   TOP 1+ ROL,   NEXT JMP, )
```

In arrays and look-up tables, not only do we require the multiplication by two but also an offset, or base address must be added. Table look-up is considerably faster if these computations are combined and performed in code. 2*+ does this.

```
CODE 2*+   ( addr N --- addr+2*N /mult by 2 & add base address     *)
           TOP ASL,  TOP 1+ ROL,  CLC,  TOP 2+ LDA,  TOP ADC,
           TOP 2+ STA,  TOP 3 + LDA,  TOP 1+ ADC,  TOP 3 + STA,
           POP JMP,   END-CODE
```

The converse of 2* is 2/ which performs an arithmetic shift right. Note that this may not be the same as signed integer division if the number is negative.

```
CODE 2/    ( U --- U/2 /does arithmetic shift right                *)
           TOP 1+ LSR,  TOP ROR,  NEXT JMP,  END-CODE
```

Alternatively, using the hand assembler definitions:

```
:CODE 2/   0156 , 0076 ,  ;NEXT
```

Powers of two, although having computational value, are also required to control specific bits of a port. Unsigned operations are necessary for this purpose and the definition of 2↑ meets this need.

```
: 2↑     ( U --- 2↑U /Arithmetic shift left 1 U times              *)
         32768  SWAP -1 DO  2*  LOOP  ;
```

Alternatively in code:

```
CODE 2↑    ( U --- 2↑U /given U [0-15], leaves 16-bit 2↑U          *)
           TOP LDA,  TOP STY,  TAY,  ( set count of shifts  left )
           SEC,  BEGIN,  TOP ROL,  TOP 1+ ROL,  CLC,  ( do shift )
                         DEY,  0<  UNTIL,
           NEXT JMP,   END-CODE
```

Without the assembler this becomes:

```
( CODE FOR 2↑U                                        RDG-831106 )
HEX
CREATE 2↑  ( U --- 2↑U   /RETURN UNSIGNED POWER OF 2           *)
         0085 , 0094 , A8 C, 38 C, 0036 , 0136 ,
         18 C, 88 C, F810 , 4C C, 0642 ,  SMUDGE
DECIMAL
(        LDA 00,X     GET LSB TO ACC
         STY 00,X     CLEAR LSB TO 0
         TAY          SET SHIFT COUNT
         SEC          GET A '1'
LOOP     ROL 00,X     INTO LOW
```

```
          ROL 01,X     DO 16-BIT SHIFT
          CLC          GET A '0' FOR REMAINING BITS
          DEY          COUNT SHIFTS
          BPL LOOP
          JMP NEXT )
```

The complement of 2↑ I have called /2↑. It is equivalent to 2↑N, but with N negative.

```
: /2↑       ( U1 U2 --- U1/2↑U2 /does U2 right shifts of U1        *)
            -DUP  IF  0 DO  2/  LOOP  THEN  ;
```

D/2↑ is a double-precision version which enables logical right shift to cross the 16-bit boundary.

```
: D/2↑      ( UD U --- UD/2↑U /LOGICAL SHIFT RIGHT U TIMES         *)
            0 DO   0  2 U/  -ROT  2 U/  SWAP DROP SWAP  LOOP  ;
```

The implementation in code for this is given below using the hand assembler definitions:

```
:CODE D/2↑ ( UD U --- UD/2↑U /DIVIDE BY 2 FOR U TIMES              *)
            00B5 , 0CF0 , 0356 , 0276 , 0576 , 0476 , 00D6 ,
            F4D0 ,  ;POP
```

Project

Write a definition to print non-destructively the contents of the return stack. Modify your definition to display additionally the corresponding name with ID. Incorporate a test to restrict printing to colon definitions only.

REFERENCES

[1] Lawrence, P. 'Recursive PICK and ROLL' *Forthwrite No. 19*, U.K. Forth Interest Group
[2] Nemeth, G. 'Within WITHIN' *Forth Dimensions*, V, No. 5, p31

3 Double Number Definitions

Some controversy exists in Forth circles as to whether the double number word extensions should be prefixed with a 2 or D. Originally the prefix was 2 because they were meant for operation on pairs of 16-bit numbers, but later opinions have mooted D for Double numbers, i.e. 32-bit. It follows therefore that you may need to change 2s to Ds according to the implementation in use. In general, standards appear to have 2s for stack manipulations and Ds for arithmetic operations, although there are no solid rules. What is preferred is that D should apply only to double-precision numbers and 2 to both double numbers and to two single-precision numbers.

Double numbers are represented by pairs of 16-bit integers, which themselves are pairs of bytes on 8-bit microprocessors. The number range spanned is $-2\,147\,483\,648 < D < +2\,147\,483\,647$. The stack sequence is: low number (hi/lo), high number (hi/lo), the high number holding the sign and being on top.

The first batch of definitions are fairly commonplace:

```
: 2SWAP     ( D1 D2 --- D2 D1 /SWAP TWO DOUBLE NUMBERS              *)
            ROT >R  ROT R>  ;
```

or alternatively:

```
            4 ROLL  4 ROLL  ;

: 2ROT      ( D1 D2 D3 --- D2 D3 D1 /DO A ROTATE WITH DOUBLES    *)
            >R >R  2SWAP  R> R>  2SWAP  ;

: 2DROP     ( D --- /DROP DOUBLE OR TWO SINGLE NUMBERS              *)
            DROP  DROP  ;

: 2DUP      ( D --- D D /DUPLICATE TOP DOUBLE NUMBER                *)
            OVER  OVER  ;

: 2OVER     ( D1 D2 --- D1 D2 D1 /COPY SECOND DOUBLE TO TOP        *)
            4 PICK  4 PICK  ;
```

If your implementation is pre-'79-standard, and PICK is missing, you could for this specific instance define 4PICK thus:

```
: 4PICK      ( COPY 4TH STACK ITEM TO TOP                        *)
             >R >R  OVER  R> SWAP  R> SWAP  ;
```

and hence define 2OVER as 4PICK 4PICK;. Defining 2OVER in this manner does, however, lead to far more stack manipulations and hence a slower version than is necessary. By rethinking the problem a more suitable alternative results:

```
: 2OVER      ( D1 D2    --- D1 D2 D1 /COPY SECOND DOUBLE TO TOP  *)
             >R >R  2DUP   R> R>  2SWAP ;
```

DMAX and DMIN are the double number equivalents of MAX and MIN. They operate on pairs of double numbers. D< is '79-standard and is defined later.

```
: DMIN       ( D1 D2 --- D3 /LEAVE MINIMUM OF TWO DOUBLE #'S     *)
             2OVER 2OVER  D<  0= IF  2SWAP  THEN  2DROP  ;

: DMAX       ( D1 D2 --- D3 /LEAVE MAXIMUM OF TWO DOUBLE #'S     *)
             2OVER 2OVER  D<     IF  2SWAP  THEN  2DROP  ;
```

3.1 DOUBLE NUMBER STORAGE

Double number variables and constants are not included in earlier standards. The definitions are straightforward and should be self-explanatory.

```
: 2@         ( ADDR --- D /FETCH DOUBLE NUMBER FROM ADDRESS      *)
             DUP 2+  @  SWAP  @  ;

: 2!         ( D ADDR --- /STORE DOUBLE NUMBER AT ADDRESS        *)
             DUP >R  !  R>  2+  !  ;

: 2CON       ( Defines 32-bit double constant                    *)
             CONSTANT , DOES>  2@  ;

: 2VAR       ( Defines 32-bit double number variable in FigForth *)
             VARIABLE ,  ;

: 2VAR       ( Defines 32-bit double number variable '79-std     *)
             VARIABLE 2 ALLOT ;
```

It may be preferable to use the names 2CONSTANT and 2VARIABLE for compatibility with later standards. The stack behaviour for 2VAR is similar to VARIABLE, i.e. early standards require the default value on the stack when the variable is defined. Versions using <BUILDS or CREATE do not usually require the default value.

3.2 DOUBLE OPERATORS

```
: D-        ( D1 D2 --- D3 /DOUBLE NUMBER SUBTRACTION            *)
            DMINUS  D+  ;

: D0=       ( D1 --- F /LEAVE TRUE FLAG IF DOUBLE NUMBER IS ZERO *)
            OR  =  ;

: D0<       ( D1 --- F /LEAVE TRUE FLAG IF DOUBLE NUMBER -VE     *)
            SWAP  DROP  0<  ;

: D=        ( D1 D2 --- F /LEAVE TRUE FLAG IF BOTH EQUAL         *)
            D-  D0=  ;

: D<        ( D1 D2 --- F /'79-STANDARD DOUBLE COMPARE           *)
            D-  D0<  ;

: D>        ( D1 D2 --- F /LEAVE TRUE FLAG IF D1 > D2            *)
            2SWAP  D<  ;

: DU<       ( UD1 UD2 --- F /COMPARE UNSIGNED DOUBLE NUMBERS     *)
            32768 +  ROT  32768 +  -ROT  D<  ;
```

or alternatively a longer, but faster version is:

```
>R >R  32768 +  R> R>  32768 +  D<  ;
```

The arithmetic shift left of 2* can be extended across a second word using the coded version D2*:

```
HEX
CREATE D2*  ( UD --- 2*UD /RETURN UNSIGNED TIMES 2               *)
            0016 , 0136 , 0236 , 0336 , 4C C, 0642 ,  SMUDGE
            ( TOP ASL,  TOP 1+ ROL,
            TOP 2+ ROL,  ROL 3 + ROL,   NEXT JMP, )
DECIMAL
```

Although D+- is a required word, it may be lacking in some implementations. DMINUS may need replacing with DNEGATE.

```
: D+-       ( D1 N --- D2 /APPLY SIGN OF N TO D1 AND LEAVE AS D2 *)
            0<  IF  DMINUS  THEN  ;
```

Finally, multiplication of two double-precision numbers need not require a quadruple-precision, or even triple-precision result since quite large numbers can be handled before overflow occurs. It may be essential to retain double-precision throughout in an iterative procedure or other looping condition and if so D* will fulfil this need.

```
: D*        ( D1 D2 --- D3 /DOUBLE NUMBER MULTIPLY               *)
            OVER  5 PICK U*  6 ROLL 4 ROLL  * +  2SWAP * +  ;
```

D* works using the relationship

$$(a+b)(c+d) = ac+ad+bc+bd$$

to calculate the result, where b and d are the high bytes. Third and fourth order terms are ignored because they represent overflow, hence bd, a fourth order term, is omitted as are the high bytes of the products ad and bc which are third order. One could check these terms beforehand and if any of them are non-zero then performing the multiplication will give rise to overflow error. Note that in Forth-'83 standard PICK and ROLL will require the preceding number to be decreased by one.

3.3 MIXED OPERATORS

Labelling conventions for double numbers were discussed, mentioning the two schools of thought. With mixed number definitions, no such thought appears to exist. There are no formal standards for mixed numbers and so combinations of U, M, D and an operator are often used with only a passing resemblance to logical naming conventions. To make matters worse, the Forth-'83 standard renames U* as UM* – oh dear!

```
 : M+        ( D1 N --- D2 /ADD SINGLE TO DOUBLE PRECISION NUMBER *)
             S->D  D+  ;
```

This simply converts the single-precision number to double and then performs a double-precision addition.

```
 : UM*       ( U D1 --- D2 /MIXED NUMBER MULTIPLICATION           *)
             >R  OVER U*  ROT  R>  *  +  ;
```

An alternative definition using a different stack sequence is:

```
 : UM*       ( D1 U --- D2 /MIXED NUMBER MULTIPLICATION           *)
             DUP ROT *  -ROT U*  ROT +  ;
```

The above two definitions work with U up to 65 535, but the resulting product must be less than 2^{31} (+2 147 483 647) or overflow will occur.

3.4 MIXED NUMBER DIVISION

The normal Forth operators for division include / and the primitive U/. Both of these leave a single-precision result, but if a large number is divided by a small number, the result is still a large number. If the numerator is double-precision, then we require a double-precision quotient.

```
 : UM/       ( D1 U --- D2 /MIXED NUMBER DIVISION                 *)
             SWAP OVER  /MOD >R  SWAP U/  SWAP DROP  R>  ;
```

UM/ works with U < 32768, but above that it depends on the numbers presented to U/ which treats values as unsigned. In '79-standard, U/ becomes U/MOD and is renamed UM/MOD in '83-standard. If you find that confusing, here is a definition for `M/MOD` which divides an unsigned double-precision number by an unsigned single-precision number, leaving a remainder `U2` and a double-precision quotient.

```
: M/MOD     ( UD1 U1 --- U2 UD2 /DIVIDE & LEAVE REM & D-QUOT     *)
            >R  0 R ( or R@ ) U/  R>  SWAP  >R  U/  R>  ;
```

The same comments regarding standards and naming conventions apply.

The double scalar word `M*/` performs one of the more useful mixed operations. It is similar to `*/` in its action, but acts on a double number using an intermediate triple-precision result. It will occur frequently in later chapters.

```
( DOUBLE SCALAR M*/                                    RDG-831214 )

: M*/      ( D1 N1 N2 --- D2 /AS FOR */ BUT OPERATES ON DOUBLES *)
           2DUP XOR  SWAP ABS >R  SWAP ABS  >R
           OVER XOR  -ROT DABS SWAP  R  U* ROT
           R> U* ROT  0 D+  R  U/ -ROT  R>  U/
           SWAP DROP  SWAP ROT  D+-  ;

```

In some Forths `R` will need to be replaced by `R@`. `-ROT` is a word previously defined which rotates the top three stack items, like `ROT`, but in the other direction.

3.5 MISCELLANEOUS

Some definitions are difficult to quantify under any specific heading and so a 'miscellaneous' heading becomes inevitable. For example, to transpose the top three stack items most readers will use `SWAP ROT` but to transpose the top four is a little more complicated. Further than this it would be worth considering a more general definition to transpose the top *N* items.

```
: 4SWAP     ( N1 N2 N3 N4 --- N4 N3 N2 N1 /TRANSPOSE TOP FOUR    *)
            SWAP  2SWAP  SWAP  ;
```

This apparent simplicity of `4SWAP` has hidden within `2SWAP` several `SWAP` and return stack moves which, if factored out in full, show a similarity to `SWAP ROT` when `ROT` is factored likewise.

Another oddment here called `A-B/A+B` (for that is what it does, brackets notwithstanding) is used in a number of mathematical functions, e.g. logarithms of large numbers. Since the function is always less than unity for positive `B`, it is scaled by 10 000.

```
: A-B/A+B   ( N1 N2 --- N3 /Compute difference/sum * 10K         *)
            2DUP  -  -ROT  +  10000 SWAP  */  ;
```

The numerator is generated by `2DUP -` and then moved by `-ROT` under the original two stack values. The addition is performed followed by the scaled division using `*/` to leave the result, times 10 000, on the stack.

Projects

1. Write a definition `D/` to complement `D*` to perform division (`D1 D2 --- D3`). What are the limitations, if any, on the values of `D1` and `D2`?
2. Rewrite the definition of `UM/` to handle a signed denominator (if you are stuck, try simplifying `M*/` by making `N1` = 1).

4 Formatting

The simplest formatting definition that I know is:

```
: U.          ( U --- /Print the top stack number as unsigned    *)
              0 D. ;
```

U. is a required word in '79-standard and later versions, but does not appear in FigForth. It prints the top stack item as unsigned, i.e. $0 \leq U < 65\,536$ and is useful for printing 16-bit addresses, either in hex or decimal. Following on from this is U.R which is similar to U. but will right-justify the number in a field width of *N.*

```
: U.R.        ( U N --- /OUTPUT U IN FIELD-WIDTH N               *)
              0 SWAP  D.R  ;
```

UD.R is the double-precision version which is totally different in its definition:

```
: UD.R        ( UD N --- /OUTPUT UD IN FIELD-WIDTH N             *)
              >R  <# #S #>  R>  OVER - SPACES  TYPE  ;
```

This operation gives the following interesting result:

```
-1. 12 CR UD.R
  4294967295 OK
```

Explanation of Forth's inherent formatting words, <# ... #> and the like, together with .R and D.R etc. are well documented in the literature. These give tremendous flexibility in the presentation of output values without too much difficulty. In this section we shall look at some particularly useful or interesting applications.

4.1 .HMS – PRINT HOURS, MINUTES AND SECONDS

The definition .HMS takes the contents of the variables MINS and SECS and formats the values in the form HH.MM.SS but SECS should be modulo 60 and MINS modulo 1440 (24 × 60). Normally control of the modulus is invested in the mechanism for incrementing or decrementing the variables.

```
SCR #116
 0 ( FORMATTING                                          RDG-840818 )
 1
```

```
0 VARIABLE MINS  0 VARIABLE SECS

: #6        ( CONVERT DIGIT AS BASE 6                         *)
            6 BASE !  #  DECIMAL  ;

: ##.       ( CONVERT TWO DIGITS AS BASE 60                   *)
            46 HOLD  #  #6  ;

: .HMS      ( --- /PRINT THE TIME AS HH.MM.SS                 *)
            MINS @ 0 <#  ##.  ##.  #>  CR TYPE
            SECS @ 0 <#  #    #6   #>  TYPE CR  ;
-->
MINS SHOULD BE MODULO 1440 FOR CORRECT OPERATION ...

```

4.2 LAT/LONG

Much of the above is similar to the formatting of angles as latitude and longitude in the form DD.MM.Mn or s and DDD.MM.Me or w from a double number representing degrees × 65536. This format enables the degrees to go up to 65535 and the minutes of arc to be resolved to one 65536th part of a degree. For astronomical calculations it may be preferable to employ minutes of arc × 65536 as a more useful scaling factor. In the examples, the appropriate compass quadrant is appended at the end.

```
SCR #117
( FORMATTING                                        RDG-840818 )

: GETDEGS  ( D --- SGN LO HI /PREPARE DEGREES                 *)
           SWAP OVER  DABS 55. D+  ;

: MINSOUT  ( U --- /FORMAT MINUTES                            *)
           46 EMIT 0  150 16384  M*/  ( TO 1/10 MINUTES )
           <#  #  ##.  #>  TYPE  ;

: ?N-S     ( F --- /OUTPUT N OR S                             *)
           IF  83  ELSE  78  THEN  EMIT  ;

: ?E-W     ( F --- /OUTPUT E OR W                             *)
           IF  87  ELSE  69  THEN  EMIT  ;

-->

SCR #118
( FORMATTING                                        RDG-840818 )

: 2NUMS    ( U --- /OUTPUT U FORMATTED AS TWO DIGITS          *)
```

```
            0  <#    #  #  #>  TYPE  ;

  : 3NUMS   ( U --- /OUTPUT U FORMATTED AS THREE DIGITS        *)
            0  <#  #  #  #  #>  TYPE  ;

  : LAT     ( D --- /DISPLAY DEGREES LATITUDE                  *)
            GETDEGS  2NUMS  MINSOUT  0<  ?N-S  ;

  : LONG    ( D --- /DISPLAY DEGREES LONGITUDE                 *)
            GETDEGS  3NUMS  MINSOUT  0<  ?E-W  ;

  : +MINS   ( U1 U2 --- D /CONVERT DEG & MINS OF ARC TO D-DEG  *)
            >R  0 SWAP  0 R>  60 U/ SWAP DROP ( rem )  0 D+  ;
```

GETDEGS extracts the sign and rounds the angle to 1/10 minute of arc. MINSOUT converts the fractional part of the degrees to tenths of minutes and then outputs the resulting number formatted with stops.

?N-S outputs N or S according to the state of the flag on the stack. Similarly with ?E-W. 2NUMS and 3NUMS print the appropriate number of digits, two for latitudes and three for longitudes.

LAT and LONG are the keywords which use the foregoing to process the double number in degrees × 65 536 and produce a formatted printout of angle.

+MINS is used to create the double number degrees from the degrees U1 and minutes U2. U1 is multiplied by 65 536 by the action of 0 SWAP and U2 is effectively multiplied as a result of the second zero and then reduced by a factor of 60. The remainder from the division is dropped and the quotient made double-precision before adding to the degrees. The polarity may then be adjusted if necessary by DMINUS or DNEGATE as appropriate.

Tenths of degrees may be suppressed by leaving out the # before ##. in MINSOUT and by changing the scaling factor. The 150 becomes 15 and in GETDEGS, 55. becomes 546. as the rounding correction. The latter figures come from 0.5 or 0.05 minutes multiplied by 65 536 and then divided by either 60 or 600 accordingly.

It may be that alternative and possibly simpler ways of achieving the same ends exist, but as part of a larger application, many words used here are common to other definitions not included. The example has been explained at some length, not because it is particularly useful, but more to illustrate the principles involved.

4.3 NAVIGATION CALCULATIONS

Now that the concept of using a double number representation for latitude and longitude has been introduced it is appropriate to look at some practical examples for navigation.

4.3.1 Range and bearing

It is at times like this that the Flat Earth Society have an advantage. Calculations relating to movements over the earth's surface invariably involve some compromise in order to yield an approximate formula. The earth is not flat; it is not even spherical. Many bodies get fatter around the waistline with age, and the earth is no exception! The reason that the earth has an equatorial diameter greater than across the poles is due to the forces of rotation acting on its mass. This difference is ignored for short distance calculations. If travel is not over the earth's surface, but in an aircraft, the earth's mean radius is increased by the aircraft's height but this effect can often be ignored.

A formula which takes into account all of the factors in the calculation requires a very complex piece of 3-D trigonometry. However, for modest distances, not only can the earth be considered spherical, but the surface of interest is approximately flat. The formula for the distance between two points on the surface can then be calculated from the latitude and longitude of the point relative to the point of interest, often one's present coordinates. The formula used is:

$$\text{Range} = 60 \times \sqrt{(\text{lat0} - \text{lat1})^2 + ((\text{long0} - \text{long1}) \cos(\text{lat0}))^2}$$

and is simply applying Pythagoras' theorem on a flat earth's surface. One degree of latitude is equivalent to 60 nautical miles, but going in the east/west direction it is reduced by the cosine of the latitude. Obviously if the north/south direction is large, the cosine will be significantly different at the two places.

Having determined the range, the other parameter required is the bearing relative to true north. If one uses the more obvious solution:

$$\text{Bearing} = \arctan\left(\frac{\text{long1} - \text{long0}}{\text{lat1} - \text{lat0}}\right)$$

there is a divide-by-zero problem if the two latitudes are the same, a not unreasonable situation. This may be avoided by using:

$$\text{Bearing} = \arccos\left(\frac{(\text{lat0} - \text{lat1}) \times 60}{\text{range}}\right)$$

The divide-by-zero problem has not gone away, but now only arises if the range is zero, which is not such a likely event.

In implementing these formulae in Forth the latitudes and longitudes are in the double-precision degrees format previously described. For a Forth definition of `ARCCOS` see Chapter 6.

```
SCR #74
 0 ( RANGE/BEARING CALCULATIONS                              RDG-841002 )
 1
 2 : NM>     ( D --- N /CONVERT DEGS * 65536 TO NAUTICAL MILES    *)
 3           15  16384  M*/  DROP  ;  ( 15 -> 150 FOR TENTH OF NM )
 4
 5 : RANGE   ( DLAT0 DLONG0 DLAT1 DLONG1 --- N /RANGE IN NM     *)
 6           2ROT D-  NM>  >R 2OVER  DCOS
 7           DUP M*  10K M/  SWAP DROP R>  ABS DUP U*
 8           ROT  10K M*/  >R >R  D- NM>   ABS DUP U*
 9           R> R>  D+  DSQRT  ;
10
11 : BEARING ( DLAT0 DLONG0 DLAT1 DLONG1 N --- D /DEGREES *65536 *)
12           >R  2ROT D-  D0<  R> SWAP >R >R ( SAVE SIGN, RANGE )
13           D- 60  R> ( RANGE ) M*/   2500 16384 M*/ ( 10K/65536 )
14           DMINUS DROP  ARCCOS   R>   ( GET SIGN FLAG )
15           IF  0 360  2SWAP  D-  THEN   ;
```

The equation holds good for distances of a few hundred nautical miles, but using integers this limits resolution to two or three figures. This can be improved by scaling by a factor of 10 or even 100. For example, in line 3, the 15 can be replaced by 150 and in line 13, the 60 by 600 to give tenths of nautical miles.

The Forth definitions are a straightforward implementation of the equation, except for scaling down the cosine by 10000 and the degrees by 65536. The expression ABS is necessary before squaring because U* would otherwise ignore the sign and treat the number as greater than 32767 and the last line is to convert the angle to the correct hemisphere. DMINUS may need replacing with DNEGATE and of course DUP U* may have been factored out.

Here are some example figures:

origin: 51°30′N 00°20′E
object: 52°30′N 01°20′E

Range = 70.68 NM Bearing = 031°54.2′

To convert degrees of arc to nautical miles at the earth's surface, 1 degree is equivalent to 60 nautical miles and if the degrees are to be in the format described, NM2DEG will perform the conversion.

```
0 ( NM TO LAT/LONG IN DEGREES * 65536 FORMAT           RDG-841002 )
1
2 : MS2DEG  ( N --- D /NM TO DEGREES LAT [LONG AT EQUATOR ONLY] *)
3           0 SWAP  ( N->D )  1 60 M*/  ;     ( 60NM = 1DEGREE )
4
```

If N represents tenths of nautical miles, then 60 in line 3 should be made 600 for correct conversion.

Project

Write another set of definitions to display time, but on this occasion from a double-precision variable containing seconds (modulo 86400). Examine the problems of extending the time cycle from 24 hours to one week.

5 The Real World

5.1 LOOKING AT A PORT

In the real world the basic computer system connects to an interface of some sort. In a situation where the computer is monitoring an input, say 8 or 16 input lines, it is useful to be able to 'see' the status of those lines as a pattern of zeros and ones. BITS formats the 16-bit binary pattern into four groups of four. It follows that for an 8-bit port this could be modified to two groups of four.

```
( BIT FORMATTING                                      RDG-841014 )

: 4#        BL HOLD  4 0 DO   #   LOOP  ;

: BITS      ( N --- /PRINT N FORMATTED AS BLOCKS OF FOUR BITS  *)
            BASE @   SWAP   2 BASE !
            0  <#    4 0 DO  4#   LOOP    #>
            CR  TYPE     BASE !  ;

: ?1BIT     ( N --- F /FLAG IS TRUE IF ONE & ONLY ONE BIT SET  *)
            DUP DUP  MINUS ( NEGATE )  AND =  ;

```

Although BITS outputs in binary, the original number base is saved and restored afterwards. A typical format would be:

```
63318 BITS
1111 0111 0101 0110  OK
```

?1BIT relies on the property of two's complement arithmetic. The least significant bit which is set, is the only bit that is set in both a number and its two's complement. Now if that bit happens to be the only bit set in the original, the two numbers are equal and ?1BIT returns a true flag.

5.2 WHICH BIT?

The purpose of LOG2.N is to convert the state of an input port to a bit number. In practice more than one bit could be simultaneously active and the arbitrary choice was to make the lowest numbered bit the one of interest.

```
( LOGARITHM TO BASE 2                                  RDG-840509 )

: LOG2.N    ( 2↑N --- N /Return bit # of lowest bit set        *)
           0 BEGIN   OVER  1 AND  0=
             WHILE   1+  >R  2/   R>
             REPEAT  SWAP DROP  ;

```

As a means of finding the logarithm of a number its usefulness is limited to rather coarse increments. However, in case this is satisfactory

$$\log_{10} N = \log_2 N \times \log_{10} 2$$

or

$$\log_2 N \times 0.301\,03$$

5.3 SCANNING TWO PORTS

Quite often the signal presented to an input port has an active low state and needs to be inverted for positive logic. The definition -@ fetches a 16-bit value and logically inverts it. When continuous monitoring of the port's address is required it is a simple matter to put it in a loop until -@ returns a non-zero value. However, life isn't always that simple and to look at two ports where the addresses are not consecutive is more complex. The definition 32@ looks at two 16-bit addresses and returns a value in the range 0 to 31.

```
SCR #111
( PORT READING DEFINITIONS                             RDG-840509 )

: -@        ( ADDR --- N /READ THE ADDRESS, INVERT CONTENTS     *)
           @  -1  XOR  ;     ( -1 = $FFFF )

: 32@       ( ADDR1 ADDR2 --- N /POLL ADDR1 & ADDR2, GIVE 0-31 *)
           0 SWAP BEGIN   DUP -@ DUP  0=
                  WHILE   DROP  SWAP  0=  ROT
                  REPEAT  LOG2.N ROT  0=  IF  16 +  THEN
           >R  2DROP  R>  ;
```

Operation of 32@ relies on alternately examining the contents of addresses 1 and 2 while both return a zero from -@. As soon as a non-zero is found the looping around ends and the contents are converted to a bit number. In order to determine which of the two addresses yielded a response, a flag is alternated between 0 and 1 and is used to shift address 1's contents to number 16 to 31.

When the loop is entered at `BEGIN` the stack contains

```
ADDR1 0 ADDR2
```

and just before `WHILE` the sequence is

```
ADDR1 0 ADDR2 (ADDR2) f
```

where `(ADDR2)` is the contents of address 2 as inverted by `-@` and `f` is an exit flag which is true if no active low input is present. If true then the contents are discarded, and the stack is shuffled to present:

```
ADDR2 1 ADDR1
```

by the time `REPEAT` is encountered, so that when returning to `BEGIN` the addresses have changed places and the address flag Boolean value is inverted (next time round `0=` will change the 1 to 0).

If `-@` returns a non-zero value the flag preceding `WHILE` will be zero and `LOG2.N` will convert the value to a number in the range 0 to 15 according to the lowest bit set. If the address flag is true then an offset of 16 is added to the number from address 2 to distinguish the signals from address 1. Finally the two addresses are dropped from the stack leaving just a number in the range 0 to 31.

5.4 HISTORY

In order to keep a record of past values of, for example, an input port, a buffer store is used. The store is required to remember the last *N* values of interest. To do this a buffer is created such that before entering a value, all previous values are moved up with the earliest being lost. The latest value is then entered. No provision has been made to initialize the buffer; it depends on its use as to what default values, if any, are required.

```
SCR #144
 0 ( HISTORY STORE                                      RDG-841210 )
 1
 2   5 CONSTANT LENGTH
 3   0 VARIABLE HISTORY  LENGTH 2 -  ALLOT
 4
 5 : C!!        ( C --- /STORE PAST CHARACTER HISTORY              *)
 6              HISTORY DUP 1-  DUP LENGTH +  1-
 7              DO  I C@  I 1+ C!  -1 +LOOP  C!  ;
 8
 9
10
11
12
13
```

C!! moves each byte stored up one, losing the oldest value, and stores the latest byte. A 16-bit word version would be similar, but with 2s instead of 1s and using 16-bit store and fetch words. Operation is straightforward, but care is needed in the definition to avoid an 'out by one' error in the store addressing. According to the standard in use define HISTORY in the form:

```
CREATE HISTORY  LENGTH ALLOT
```

5.5 LABELLING THE 6522 VIA CHIP

The processor dependent software in this book has been orientated towards users of the 6502 processor and a commonly used peripheral chip with this processor is the 6522 Versatile Interface Adapter (VIA). Control of bits in the 6522 registers using Forth can lead to some fairly unreadable statements. For example, rather than writing:

```
59437  C@ 127 AND  59437 C!
```

a more readable version would be

```
2VIA IFR 7 BIT-OFF
```

This may be realized by assigning a name to the VIA base address and suitably naming the addresses of the internal registers as in the 6522 data sheet and defining them as offsets.

```
SCR #149.
 ( 6522 VIA CHIP CONTROL DEFINITIONS                    RDG-840523 )

 $E800 CONSTANT 1VIA
 $E820 CONSTANT 2VIA

 ( ADDR1 --- ADDR2 /ADD OFFSET FOR VIA REGISTER                    *)
 : ORB     ( $0F + ) ;        : ORA      $01 +  ;
 : DDRB      $02 +  ;         : DDRA     $03 +  ;
 : T1C-L     $04 +  ;         : T1C-H    $05 +  ;
 : T1L-L     $06 +  ;         : T1L-H    $07 +  ;
 : T2C-L     $08 +  ;         : T2C-H    $09 +  ;
 : SR        $0A +  ;         : ACR      $0B +  ;
 : PCR       $0C +  ;         : IFR      $0D +  ;
 : IER       $0E +  ;


```

The ORB register appears as two discrete addresses, one with zero offset, the other with $0F offset. If you need further details of the VIA and its operation, you should refer to the data sheet since the object here is to explain a defining technique for program readability.

After the register addresses come the individual bits within those registers.

Four words are defined which yield an individual bit's status, set or clear a specified bit, or toggle a specified bit. The bit number ranges 0 to 7 as defined, but may be readily changed to 0 to 15 as required.

```
SCR #148
( BIT-MANIPULATION, C = 0 TO 7                          RDG-830405 )

: BIT?      ( C ADDR --- F /1 = ON   0 = OFF                      *)
            C@  SWAP  2↑  AND  0=  0=  ;

: BIT-ON    ( C ADDR --- /SET SPECIFIED BIT ON                    *)
            DUP >R  C@ SWAP 2↑       OR      R>  C!  ;

: BIT-OFF   ( C ADDR --- /TURN SPECIFIED BIT OFF                  *)
            DUP >R  C@ SWAP 2↑  255 XOR AND  R>  C!  ;

: BIT-TOGGLE  ( C ADDR --- /TOGGLE SPECIFIED BIT                  *)
            SWAP 2↑   TOGGLE  ;
```

The word 2↑ may be high-level or code as desired. We may now turn off bit 7 of the interrupt flag register of the second VIA chip with `2VIA IFR 7 BIT-OFF` as before.

Project

Define a word `SEE.HISTORY` to reveal the current byte contents of the 'history' store. Test it thoroughly to avoid an 'out by one' error. Write the equivalent definitions for 16-bit data. What are the problems in identifying initial garbage from valid data?

6 Mathematics

6.1 TRIGONOMETRIC FUNCTIONS

The philosophy that is adopted here for trigonometric functions is that for any given application, only one such function is required. As a result that function must as far as possible be a 'stand-alone' definition which does not depend on other trigonometric functions being present for its operation. There are a number of ways of implementing 'trig' functions in Forth and each has its particular advantages and disadvantages. Assuming that a floating point package is not available, it comes down to making the best use of integer arithmetic. The final choice is from a trade off between speed, memory space and resolution.

Where resolution is required, it is not sufficient to use degrees of arc alone. Here the previously mentioned method of employing a double number is used. The high digit is still degrees, but times 65 536. The low order digit is the fractional part, although this too is times 65 536. To convert this format back to signed single-number format you should use D->S:

```
: D->S      ( D --- N /CONVERT D-DEGREES TO N WITH ROUNDING        *)
            32768 0 D+  SWAP DROP  ;
```

When using sines and cosines the result obtained is within the range plus or minus one. To overcome this fractional problem a scaling factor must be introduced: typically 16384 and 10000. The former gives slightly more accuracy, while the latter gives a more readable result and is used here.

6.1.1 Sine function

The usual method of producing a sine function in Forth is by look-up table. This has the advantage of speed, but at the expense of memory. The finer the increments, the more storage is required. An alternative is to evaluate a series. This is the technique used by Bumgarner [1].

```
( SCALED INTEGER SIN FUNCTION                         JOB-82MAR31 )

10000 CONSTANT 10K     ( THE SCALING CONSTANT )
    0 VARIABLE XS      ( THE SQUARE OF THE SCALED ANGLE )

: KN    ( A B --- M /M=10000-AX*X/B ..A COMMON TERM IN SERIES *)
```

```
        XS @  SWAP / MINUS  10K */ 10K +  ;

: (SIN) ( THETA --- 10K*SIN /-15708<THETA<15708 RADIANS * 10K *)
        DUP 10K */ XS ! ( save x↑2 )    10K  ( start series )
        72 KN  42 KN  20 KN  6 KN       10K */  ( times x )  ;

: SIN   ( THETA --- 10K*SIN / 0 - 90 DEGREES ONLY             *)
        17453 100 */  (SIN)  ;  ( DEG TO RADIANS*10K)
;S
SIN(X) = X*(1-X↑2/6 (1-X↑2/20 (1-X↑2/42 (1-X↑2/72 )))) APPROX.
```

Depending on your implementation, `MINUS` may need replacing with `NEGATE` ('79-standard and later) and `VARIABLE` will not need the preceding zero.

6.1.2 Cosine function

The usual method of producing a cosine function is to convert the angle so that the sine table can be used. The expression is

$\cos(A) = \sin(90 - A)$

If the cosine function is all that is required, it may be obtained by using a series which has similarities to that used for sine.

```
( SCALED INTEGER COS FUNCTION - after J.O.B.         RDG-840525 )

10000 CONSTANT 10K    ( THE SCALING CONSTANT )
    0 VARIABLE XS     ( THE SQUARE OF THE SCALED ANGLE)

: KN    ( A B --- M /M=10000-AX*X/B ..A COMMON TERM IN SERIES *)
        XS @  SWAP / MINUS  10K */  10K +  ;

: (COS) ( THETA --- 10K*COS /-15708<THETA<15708 RADIANS * 10K *)
        DUP 10K */ XS ! ( SAVE X↑2 )
        10K ( Start series )      56 KN  30 KN  12 KN  2 KN  ;

: [COS] ( THETA --- 10K*COS / 0 - 90 DEGREES ONLY             *)
        17453 100 */  (COS)  ;  ( DEG TO RADIANS*10K )
-->
COS(X) = (1-X↑2/2 (1-X↑2/12 (1-X↑2/30 (1-X↑2/56 )))) APPROX.
```

6.1.3 DCOS

If the angle is presented in the format degrees × 65 536, then `DCOS` will return the cosine × 10000.

```
( DCOS - DOUBLE COSINE                                RDG-840606 )

: COS   ( N --- 10K*COS /GIVES COS FOR ALL DEGREES             *)
```

```
        XS @  SWAP / MINUS  10K */ 10K +  ;

: (SIN) ( THETA --- 10K*SIN /-15708<THETA<15708 RADIANS * 10K *)
        DUP 10K */ XS ! ( save x↑2 )    10K  ( start series )
        72 KN  42 KN  20 KN  6 KN       10K */  ( times x )  ;

: SIN   ( THETA --- 10K*SIN / 0 - 90 DEGREES ONLY              *)
        17453 100 */  (SIN)  ;  ( DEG TO RADIANS*10K)
;S
SIN(X) = X*(1-X↑2/6 (1-X↑2/20 (1-X↑2/42 (1-X↑2/72 )))) APPROX.
```

Depending on your implementation, `MINUS` may need replacing with `NEGATE` ('79-standard and later) and `VARIABLE` will not need the preceding zero.

6.1.2 Cosine function

The usual method of producing a cosine function is to convert the angle so that the sine table can be used. The expression is

$$\cos(A) = \sin(90-A)$$

If the cosine function is all that is required, it may be obtained by using a series which has similarities to that used for sine.

```
( SCALED INTEGER COS FUNCTION - after J.O.B.        RDG-840525 )

10000 CONSTANT 10K     ( THE SCALING CONSTANT )
    0 VARIABLE XS      ( THE SQUARE OF THE SCALED ANGLE)

: KN    ( A B --- M /M=10000-AX*X/B ..A COMMON TERM IN SERIES *)
        XS @  SWAP / MINUS  10K */  10K +  ;

: (COS) ( THETA --- 10K*COS /-15708<THETA<15708 RADIANS * 10K *)
        DUP 10K */ XS ! ( SAVE X↑2 )
        10K ( Start series )     56 KN  30 KN  12 KN  2 KN  ;

: [COS] ( THETA --- 10K*COS / 0 - 90 DEGREES ONLY              *)
        17453 100 */  (COS)  ;  ( DEG TO RADIANS*10K )
-->
COS(X) = (1-X↑2/2 (1-X↑2/12 (1-X↑2/30 (1-X↑2/56 )))) APPROX.
```

6.1.3 DCOS

If the angle is presented in the format degrees × 65 536, then `DCOS` will return the cosine × 10 000.

```
( DCOS - DOUBLE COSINE                                  RDG-840606 )

: COS   ( N --- 10K*COS /GIVES COS FOR ALL DEGREES             *)
```

```
 3         360 MOD  ABS  DUP 270 > IF  360 -  THEN   DUP
 4         90 > IF  180 SWAP - [COS] MINUS  ELSE  [COS]  THEN  ;
 5
 6 : [DCOS] ( D --- 10K*COS /GIVES COSINE FOR DEGREES * 65536    *)
 7         20 7510 M*/  DROP  (COS)  ;
 8
 9 : DCOS   ( D --- 10K*COS /GIVES COSINE FOR DEGREES * 65536    *)
10         360 MOD  DABS DUP 270 > IF  0 360  D-  THEN  DUP
11         90 > IF    0 180  2SDWAP D-  [DCOS]  MINUS
12         ELSE  [DCOS]  THEN  ;
13
14
15
```

The fraction 20/7510 is a simplification of $\pi/180/65536$ scaled by 10K. Do not reduce this to 2/751 or there will be a loss of accuracy. Most errors arise from the division in KN causing truncation and then the summing of a number of such errors. It may be possible to squeeze some improvement by multiplying the KN coefficients by 10 and using ROUND 10 / after the division. As it stands the error is less than 2 in 10^4.

Since DCOS uses the 'infinite' series of (COS), the look-up table alternative may be preferred. This is based on the expansion:

$$\cos(A + B) = \cos(A)\cos(B) - \sin(A)\sin(B)$$

If we make A = degrees and B = minutes, then B is less than one degree and the following approximations may be used: $\cos(B) \rightarrow 1$ and $\sin(B) \rightarrow B$. Therefore we can say

$$\cos(A + B) = \cos(A) - B\sin(A)$$

where B is in radians, hence the second definition of DCOS:

```
0 ( DOUBLE NUMBER COSINE                             RDG-840927 )
1
2 : DCOS   ( D --- N /Gives COS * 10K for degs * 65536          *)
3         DUP COS  -ROT SIN  0 SWAP
4         720 M*/  31416 10000 M*/  1  16384 M*/  DROP  -  ;
```

The numbers in line 4 arise from the conversion of the angle in double degrees format to radians, i.e. $\pi/180/65536$.

Both the sine and cosine series could be evaluated using the definitions 1+DX and 1+SX which are defined later for the arctan function.

6.1.4 Quick and dirty methods

Suppose we sum an infinite series to, say, 10 or 12 terms we may find that it involves terms that no longer contribute anything to accuracy, and yet can take a lot of unnecessary time. If the series is aborted after only a few terms a lot of accuracy is lost, but if the coefficients of the series are adjusted, we can improve the accuracy without extensive computation. Normally SIN and COS functions are expressed in terms of Maclaurin's series, but Chebyshev polynomials can be used to give a 'best fit' over a limited range of values. For sin(A) where A is in degrees the following approximation may be used:

$$\sin(A) - \frac{63A}{3240} - \frac{A^2}{10800}$$

This expression is easy to implement in Forth and is simply:

```
( SIN OF ANGLE BY CHEBYSHEV                            RDG-841103 )

        10000 CONSTANT 10K

: SIN   ( N1 --- N2 /RETURN SIN*10K APPROX                      *)
        DUP  DUP U*   10K 10800  M*/  DMINUS
        ROT 1000 U*  630  3240  M*/  D+  DROP  ;
```

So what is the catch? Firstly, the expression is only valid for $0 \leqslant A \leqslant 90$ and secondly it is only correct at 0, 30 and 90 degrees. For intermediate values, the second digit may be out by one or two. Typical error figures are

$30 < A < 90$	less than 3.8%
$0 < A < 30$	approaching −11.4% as $A \to 0$

Another, more complex expression, called a rational polynomial approximation, gives much better accuracy and requires the angle in radians:

$$\sin(X) = \frac{X - 7X^3/60}{1 + X^2/20}$$

and this too may be readily implemented in Forth by:

```
SCR #130
( SIN[X] BY RATIONAL POLYNOMIAL                        RDG-841103 )

                 10000 CONSTANT 10K

: SIN   ( N1 --- N2 /RETURN SINE OF N1 RADIANS - N2 IS *10K   *)
        20000 OVER DUP U*    7 30 M*/  10K U/
        SWAP 5000 > + ( round )  -
               OVER DUP U*    1 10 M*/  10K U/
        SWAP 5000 > + ( round )  20000 +
        0 -ROT ( make N double )  M*/  DROP  ;
```

This version has been scaled to reduce loss of accuracy by truncation and to avoid resorting to triple-precision division and other complexities. The outcome is a word which gives a result within one digit of the values obtained using the algorithm with floating point arithmetic. The error is greatest at 90 degrees (1.5708 radians) where the result is:

```
15708 SIN . 9958 OK
```

The expression is scaled by 20 000 and X (radians) by 10 000. The expression `10K U/ SWAP 5000 > +` adds one (rounds) the quotient if the remainder is more than half the denominator (10000). This is rather naughty because it assumes that the true flag from the comparison will be +1 whereas in many systems it is −1. To be portable the `+` should be replaced by `IF 1+ THEN` in both occurrences of the expression. Some factoring of the definition could be made, but little is gained in so doing in this instance unless the factored definitions can be used in further expressions.

6.2 INVERSE FUNCTIONS

The series for inverse functions are somewhat more complex to evaluate and you may prefer to use a successive approximation technique, i.e. 'trial and error'. The expression trial and error is used advisedly since errors will arise from the error in the expression to evaluate the sine/cosine/tangent of each trial, plus the difference between the final try and the actual submitted value before 'calling it a day'. This is related to the smallest number that can be resolved. In addition there may be computational errors.

6.2.1 Arccos

```
( ARCCOS BY SUCCESSIVE APPROXIMATION                 RDG-840930 )

: ARCCOS   ( N --- D /CONVERT TOS [0-10K] TO DEGREES * 65536  *)
        0 90  ROT  0 90 ROT 23 0
        DO      >R
                2SWAP  1 2 M*/  2SWAP
                2DUP  DCOS  R <
                >R   2OVER   R>
                IF  DMINUS  THEN  D+  R>
```

```
 9          LOOP    DROP 2SWAP  DROP  DROP  ;
10
```

ARCCOS sets up two double-precision numbers on the stack, both equivalent to 90 degrees with N manipulated to the top. The DO...LOOP is then set up for 23 iterations. Line 5 divides the second double number by two. This is then added to or subtracted from the top double number depending on whether the top item produces a DCOS value which is too high or low. The flag for this comparison is stored on the return stack at line 7 and line 8 decides whether to add or subtract the second double number. On leaving the LOOP the stack is then tidied to leave the result.

The routine requires DCOS to be defined first, which goes against the previously defined objectives. It is also slow because of DCOS in the loop, and there are errors due to M*/ always rounding down through the closing iterations. Don't be alarmed by all these errors; they are not huge, and the accuracy may be sufficient. You may wonder why there are 23 iterations through the loop, why not 32? Since the largest number is 90 degrees (i.e. 90 × 65 536) after 23 right-shifts (128 × 65 536), there is nothing left! Finally, as before, DMINUS may need to be replaced by DNEGATE.

6.2.2 Arctan

The series to evaluate arctan (X) is given by:

$$\arctan(X) = X - \frac{X^3}{3} + \frac{X^5}{5} - \frac{X^7}{7} + \ldots.$$

and is the one used in the next example. However, as an infinite series, it has a serious limitation in that it is slow to converge.

For this reason ATN has been arranged to compute as many terms as are required with a limit of 50. This can be raised as desired if you don't mind waiting for the answer. In fact for $X \geqslant 1$ the series does not converge at all!

As an example of this shortcoming, arctan (0.5) = 0.46365 and 5000 ATN leaves 4637 after summing 6 terms. 9000 ATN requires 26 terms to give 7328, but 9600 ATN requires 50 terms and 10000 ATN requires well over 500 terms. Above this, forget it, you're heading towards infinity and integer arithmetic cannot cope! It is included here as an academic exercise.

```
0 ( ATN BY SERIES                                          RDG-840916 )
1
2  0 VARIABLE (ATN)
3
4 : ATN    ( N1 --- N2 /CONVERT STK TOP TO ANGLE IN RADIANS        *)
5          (ATN) ! DUP DUP 10K */ XS ! ( save X↑2 )  1 ( coefft )
6          51 1 DO OVER OVER /  DUP (ATN) +! 0= IF  LEAVE  THEN
7                SWAP XS @ 10K */  SWAP
8                DUP ABS 2+  SWAP MINUS +-
9             LOOP DROP DROP  (ATN) @  ;
```

ATN uses the variable (ATN) to accumulate terms in the series until either the next term is insignificant or 50 terms have been summed. The manipulations in line 8 are to cause each successive term to alternate in sign.

ATN uses VARIABLE and MINUS which have been mentioned earlier, but LEAVE may act differently in more recent Forth standards. It should have little effect on ATN's operation.

6.2.3 Arctan by successive approximation

The problem here is that the method requires TAN which must be defined, either in terms of SIN and COS, or as a series. Since the philosophy of this chapter is to have each function as a stand alone item, to use both sine and cosine functions is inadmissible. If TAN is to be defined using a series simply to obtain the arctangent then we might as well use an appropriate series for arctan directly.

6.2.4 Arctan by rational polynomial approximation

Another expression which approximates to arctan(X) is the rational polynomial (Pade approximation) thus:

$$\arctan(X) = \frac{X + \frac{7}{9}X^3 + \frac{64}{945}X^5}{1 + \frac{10}{9}X^2 + \frac{5}{21}X^4}$$

where again $X < 1$. To implement this using Forth's integer arithmetic, we must again scale and factorize the expression.

```
SCR #127
 ( ARCTAN BY RATIONAL FRACTION                          RDG-841202 )

 : 3PICK   6 SP@ + @  ;       ( 'Cos I don't have PICK )

 : 1+DX    ( X D1 N1 N2 --- X D2 /CALCULATE NEXT TERM            *)
           M*/   10K 0 D+  3PICK 10K M*/  ;

 : 1+SX    ( X D1 N1 N2 --- N3 /CALCULATE FINAL TERM             *)
           M*/   10K 0 D+   DROP SWAP DROP  ;

 : ATN     ( N1 --- N2 /RETURN ARCTAN * 10K     -1 < X < +1      *)
           DUP  DUP M*    10K M/  SWAP DROP  DUP S->D
           3PICK 3PICK 3PICK  3 14 1+DX  10 9 1+SX
           >R 64 735 1+DX  7 9 1+SX M* R>  M/ SWAP DROP  ;


```

Accuracy is surprisingly good considering the 'swap drops' and the number of divisions. The error in the equation is greatest when $X = 1$ and increases at lower values for the calculation in Forth. Table 6.1 gives some sample figures.

Table 6.1 Accuracy of arctan figures obtained from rational polynomial approximation.

X	*arctan*	*equation*	*Forth*	*approx. error*
10000	0.785398	7855.86	7855	.013%
8423	0.700007	7000.53	7000	.001%
6842	0.600043	6000.51	6000	.007%
3094	0.300058	3000.58	3000	.019%

The figures are the same for positive or negative values and the greatest error is some 0.013%. The problem is knowing how accurate the 'correct' figures are: they come from a computer using floating point calculation of another approximation to find `ATN(X)`!

6.2.5 Arcsin

The usual series for arcsin is derived from an integrated binomial expansion of the first derivative, but then you already knew that, didn't you? In short the series runs thus:

$$\arcsin(X) = X + \frac{1}{6}X^3 + \frac{3}{40}X^5 + \frac{5}{112}X^7 + \frac{35}{1152}X^9 + \ldots$$

which can be factored out to repeated sum and product terms so that we can reuse the definitions for `ATN`, but with different coefficients. Like a lot of infinite series, the end result is only accurate for an infinite number of terms. Happily in practice a lesser number of terms can be used with some sacrifice in precision. With arcsin(X) sufficient accuracy may be obtained using only the first four terms above, although five are included in `ARCSIN` as defined.

```
SCR #124
 ( ARCSIN BY SERIES                                   RDG-841202 )

 : 3PICK    6 SP@ + @  ;

 : 1+DX     ( X D1 N1 N2 --- X D2 /CALCULATE NEXT TERM          *)
            M*/   10K 0 D+  3PICK   10K M*/  ;

 : 1+SX     ( X D1 N1 N2 --- N3 /CALCULATE FINAL TERM           *)
            M*/   10K 0 D+   DROP SWAP DROP  ;

 : ARCSIN   ( N1 --- N2 /RETURN ARCSIN * 10K      -1 < X < +1   *)
            DUP  DUP M*   10K M/  SWAP DROP  DUP S->D
            8757 17280 1+DX  600 10008 1+DX  18 40 1+DX  1 6 1+SX
            M* 10K M/ SWAP DROP  ;


```

The fifth term has little effect if X is small, but as X approaches 1 (10000 when scaled), the series converges more slowly and the extra term may be considered useful. The degree of accuracy obtained is shown in Table 6.2.

Table 6.2 Accuracy of arctan figures obtained by series.

X	*arctan*	*series*	*Forth*	*error*
1.0000	1.570796	1.24616	12461	20%
0.9000	1.119769	1.06794	10678	5.18%
0.8000	0.927295	0.910852	9108	1.76%
0.7000	0.775398	0.770142	7700	0.67%
0.6000	0.643501	0.641958	6419	0.25%
0.5000	0.523599	0.523212	5232	

If greater accuracy is required as X approaches 1 perhaps the method of look-up table should be used, especially since the Nth term becomes increasingly difficult to determine. Alternatively one could 'adjust' the coefficients of the higher order terms.

6.3 POWERS

It is not difficult to write a routine to raise a number X to a power N. One simply multiplies X by X for $N-1$ times. The following definition ** does just that, but filters out the special cases where N is one or zero. Negative values are not catered for.

```
SCR #122
 0 ( ** - PERFORM X↑N BY REPEATED MULTIPLICATION       RDG-841117 )
 1
 2 : **     ( X N --- X↑N /RAISE X TO POWER N - N POSITIVE        *)
 3          DUP 0=  IF     DROP DROP 1
 4                  ELSE   DUP 1 =
 5                         IF     DROP
 6                         ELSE   OVER SWAP 1 -
 7                                0 DO OVER *  LOOP
 8                                SWAP DROP
 9                         THEN
10                  THEN  ;
11
12
13
14
15
```

Line 3 handles the case where $N=0$ by clearing X and N from the stack and leaving the known result of 1. Lines 4 and 5 handle $N=1$ by dropping N and leaving X. Line 6 sets up the parameters for a DO. . .LOOP to give the correct number of times to multiply by X. Line 8 then drops X to leave X^N.

This method can be rather limiting. Firstly X and N need only be fairly small before overflow occurs, e.g. 10^4 and 2^{15} are obviously near the limit. Secondly having multiplied X by X to get X squared, why not obtain X^4 by squaring X squared? By generating terms in X, X^2, X^4, X^8 and including them as required in the product according to the bit pattern of N when expressed in binary form, the number of multiplications is minimized [2]. Screen 120 shows the principle and D** is a double number version. Try 10^9 or 2^{31} for example.

```
SCR #120
 0 ( X**N - EXPONENTIATION                                 RDG-841021 )
 1
 2 : **     ( X N --- X**N /RAISE X TO POWER N                       *)
 3          >R  1 SWAP
 4          BEGIN   R 1 AND   IF  SWAP OVER * SWAP  THEN
 5                  R>  2/  -DUP
 6          WHILE   >R  DUP *
 7          REPEAT  DROP  ;
 8
 9 : D**    ( N1 N2 --- D /RAISE X TO POWER N DOUBLE # RESULT        *)
10          >R  1. ROT 0
11          BEGIN   R 1 AND   IF  2SWAP 2OVER D* 2SWAP THEN
12                  R>  2/  -DUP
13          WHILE   >R 2DUP D*
14          REPEAT  2DROP  ;
15
```

The definition of 2/ could readily be in code in applications where speed is important. R is the FigForth word to fetch the top number on the return stack. This is called R@ in some Forths. The definition D* multiplies two double-precision numbers and leaves a double-precision result. A suitable definition appears in Chapter 3.

6.4 EXP(X) OR e^X

The conventional series for e^X is an infinite series which is not a suitable choice to use in Forth. A shortened series for e^X is given by:

$$e^X = 1 + \frac{383}{384}X + \frac{1}{2}X^2 + \frac{17}{96}X^3 + \frac{1}{24}X^4$$

In Forth it can be realized by the following, where X and e^X are both scaled by 10000.

```
SCR #129
 0 ( EXPONENTIAL FUNCTION                                  RDG-841103 )
 1
```

```
: 3PICK          6 SP@ + @  ;

: 1+DX  ( X D1 N1 N2 --- X D2 /CALCULATE NEXT TERM            *)
        M*/   10K 0 D+  3PICK 10K M*/  ;

: 1+SX  ( X D1 N1 N2 --- N3 /CALCULATE FINAL TERM             *)
        M*/   10K 0 D+   DROP SWAP DROP  ;

: EXP   ( N1 --- N2 /RETURN EXPONENTIAL* 10K -1 < X < +1      *)
        DUP   S->D   4 17 1+DX  17 48 1+DX  192 383 1+DX
        383 384 1+SX  ;

( EXP[X] = 1 + 383/384.X.[1+192/383.X.[1+17/48.X.[1+4/17.X]]] )
```

The definition of EXP is not unlike ATN in that it uses the same building blocks, but like ARCSIN is a lot simpler because only one polynomial is involved. The series is not infinite, but is an approximation valid for ABS(X) less than 1. See Table 6.3.

Table 6.3 Values of e^X

X	e^X	*series*	*Forth*
1.0000	2.718282	27161.5	27160
0.1	1.10517	11049.2	11048
0	1.000000	10000	10000
−0.1	0.904837	9050.88	9052
−1.0000	0.367879	3671.88	3672

For a treatment of logarithms in Forth see reference [3].

6.5 BINOMIALS

The binomial series gives the value of numbers near to unity raised to a power. The form of the series suitable for expressing in Forth is:

$$(1+X)^k = 1 + kX.\left(\frac{X(k-1)}{2}.\left(1+\frac{X(k-2)}{3}.\left(1+\frac{X(k-3)}{4}.\left(\ldots\right)\right)\right)\right)$$

where $-1 < X < +1$.

The behaviour of the series varies according to the value of k as X approaches its permissible limits. Generally accuracy deteriorates because the series fails to converge to a sensible value with a modest number of terms. If X is near to -1 the expression behaves badly with negative powers.

```
SCR #131
( BINOMIAL FUNCTIONS                                   RDG-841103 )

: (1+X).5         ( N1 --- N2 /RETURN BINOMIAL*10K -1 < X < +1  *)
                  DUP   S->D   -7 10 1+DX  -5 8 1+DX  -1 2 1+DX
                  -1 4 1+DX     1 2  1+SX  ;

: 1/(1+X).5       ( N1 --- N2 /RETURN BINOMIAL*10K -1 < X < +1  *)
                  DUP   S->D   -9 10 1+DX  -7 8 1+DX  -5 6 1+DX
                  -3 4 1+DX    -1 2  1+SX  ;
```

The polynomial is evaluated as before using the appropriate coefficients, the only difficulty being to find a suitable name for each word! See Tables 6.4(a) and (b).

(a) **Table 6.4** Behaviour of the binomial series.

X	$(1+X)^{1/2}$	*series*	*Forth*
1.0	1.414214	1.42588	14258
0.5	1.22475	1.22412	12249
0	1.000000	1.00000	10000
−0.5	0.707107	0.707642	7077
−0.75	0.500000	0.509472	5095

(b)

X	$1/(1+X)^{1/2}$	*series*	*Forth*
1.0	0.707107	0.589811	5899
0.5	0.814087	0.814087	8141
0	1.000000	1.000000	10000
−0.75	2.000000	1.86269	18626

6.6 SQUARE AND CUBE ROOTS

Various methods exist to calculate the square root of a number, at least one of these being introduced in one's school days. However, some methods are more amenable to computer implementation than others, particularly where integer arithmetic is concerned.

6.6.1 Square roots

The first version here uses a successive approximation technique to produce a square root value of a 16-bit integer (actually 15-bits since it must be a positive number). It uses the equation:

$$X' = \frac{\left(\frac{N}{X} + X\right)}{2}$$

to give a second approximation X' to the square root of N from a first approximation X. Mathematicians will recognize this equation as the Newton-Raphson method. The first approximation chosen in the example is purely arbitrary and the number of iterations used is adequate. No mathematical or scientific basis has been used to opitimize them except to test that they suffice.

```
( SQUARE ROOT 16-BIT                                   RDG-840804 )

: APPROX  ( N X --- N X' /COMPUTER NEXT APPROXIMATION          *)
          OVER OVER  / +  2 /  ;

: SQRT    ( N1 --- N2 /RETURN SQUARE ROOT OF N  [32767 MAX]    *)
          60  5 0 DO  APPROX  LOOP  SWAP DROP  ;

: 3DUP    ( N1 N2 N3 --- N1 N2 N3 N1 N2 N3 /DUP TOP 3 ITEMS    *)
          ( = 3 PICK  3 PICK  3 PICK )
          >R OVER R SWAP >R OVER R> SWAP R>  ;
-->.




```

The number that we wish to find the square root of is often the result of the product of two numbers, for example the geometric mean. It follows therefore that it is likely to be a double number, hence `DSQRT`:

```
( SQUARE ROOT OF 32-BIT DOUBLE NUMBER                  RDG-840804 )

: DSQRT   ( D --- N /RETURN SQUARE ROOT OF DOUBLE NUMBER D     *)
          127  7 0 DO 3DUP
                   M/MOD ROT DROP  ROT 0 D+  2 U/
                   SWAP DROP
                LOOP  >R DROP DROP R>  ;






```

DSQRT works with the full 32-bit unsigned value. Since some implementations do not have PICK, the author's included, it has been coded without. The definition of 3DUP is equivalent to 3 PICK 3 PICK 3 PICK and duplicates the top three stack items. 2DROP may be used instead of DROP DROP. ('83-standard has PICK and for this 3 should be 2.)

6.6.2 Cube roots

The process of finding the cube root is similar to that used for square roots but the second approximation X' is given by:

$$X' = \frac{\left(\frac{N}{3X^2} + 2X\right)}{3}$$

Newton's method is in fact a general method for the Nth root of a number and this is the specific approximation for the cube root. Without a floating point arithmetic package, going beyond the cube root has little value (the fourth root of a double-precision integer can only have at best 3-digit resolution). Even for the cube root it should be fairly obvious that if N is only a 16-bit value, the cube root is less than 32 and is of only 1 or 2 significant figures, not a lot of use. Ideally, N should be a triple-precision number to yield a single-precision result. The more dedicated among you may be inclined to write a triple number version, but for the less ambitious the double number version will theoretically handle numbers up to 277^3, but in practice this becomes $168^3 = 4787099$ because of liberties taken in calculating X'. In particular, U* produces a double number result which is not utilized. This is to avoid performing a division with two double numbers.

In CUBEROOT it has been attempted to optimize the number of iterations and the first guess. The optimum first guess is, of course, the correct result, but since this is unknown, a value has been chosen which is a compromise between 1 and 168. If CUBEROOT is to be used for a smaller range of results, then the first guess could be changed and the number of iterations reduced.

```
( CUBEROOT                                         RDG-841029 )

: (CUBE)  ( D N1 --- D N2 /DERIVE APPROXIMATION N2 FROM N1    *)
          3DUP   DUP U* DROP ( hi ) M/MOD  ROT DROP ( rem )
          ROT 2*  0 D+   3 U/  SWAP DROP  ;

: CUBEROOT ( UD --- N /LEAVE CUBE ROOT OF UD                  *)
          84  10 0 DO   (CUBE)   LOOP
          SWAP DROP SWAP DROP  ;
```

`(CUBE)` is the inner routine to derive a new approximation from the previous. `3DUP` replicates the top three stack items, i.e. the double number and single-precision approximation, the remainder of the definition is the straightforward evaluation of the equation.

`CUBEROOT` uses an initial guess of 84. Even if as a first guess this value is wildly out, 10 iterations are sufficient to handle the worst case.

6.7 POLAR COORDINATES

As an example of using the double-precision square root, conversion from cartesian to polar coordinates is given by:

$$\text{Modulus } R = \sqrt{(X^2 + Y^2)} \text{ and argument } \theta = \arctan\ (Y/X)$$

In Forth the modulus is simply defined by:

```
: MODULUS     ( N1 N2 --- N3 /Do SQRT of sum of squares          *)
              DUP  U*  ROT  DUP  U*  D+  DSQRT  ;
```

In `MODULUS` the expression `DUP U*` simply squares the top stack item each time and the two respective double-precision results are added before being presented to `DSQRT`. It is possible to extend the concept to calculate rms (root-mean-square) values for N items. If squares are likely to be commonplace in an application, then it will be worth factoring out `DUP U*` into a separate definition. It could then be included in the definition to find the area of a circle given the radius. As with any limited precision calculation, the radius may need to be scaled before the calculation is performed. The ratio 355/113 is a close approximation to π.

```
: AREA        ( N --- D /GIVES AREA OF CIRCLE FROM RADIUS         *)
              DUP U*  355 113 M*/  ;
```

6.8 MODVARIABLE

The range of values of a variable can be constrained using the word `MOD` as, for example, when an index to an array is to be constrained to the array limits. If several arrays or memory segments are to be indexed, it can be an advantage to define a family of such words. `MODVAR` is a defining word which will do this.

```
( MODVARIABLE                                          RDG-840703 )

: MODVAR  ( N1 N2 --- /VARIES AS MOD[N2] + N1                    *)
          ( N3 <name> --- N4 )
          <BUILDS   [COMPILE] LITERAL ,
                    [COMPILE] LITERAL ,
          DOES>     DUP @  ROT SWAP MOD SWAP 2+ @ +  ;
```

```
8 ( eg: 3 7 MODVAR INDEX )
9
```

The action of MODVAR is to constrain the function so defined (INDEX in the example) such that N INDEX varies between 3 + (0 to 6) as N varies over (0 to 6) + 7 *K* where *K* is an integer. You may wish to experiment with the action of replacing MOD in line 6 with <MOD> as defined earlier in Chapter 2.

You may recall that <BUILDS is the FigForth equivalent of CREATE and determines what compiling action will occur when words defined by MODVAR are compiled into the dictionary. MODVAR works by twice compiling the cfa of LITERAL followed by the top stack item during the <BUILDS action. When a word defined by MODVAR, such as INDEX, is executed it uses these two numbers at run time in accordance with the actions following DOES> to determine the resulting value left by INDEX.

Projects

1. What are the problems in producing a definition to evaluate log(X) in Forth?
2. Write and test a definition for exp(X) scaled by 10000 using the approximation:

$$e^X = \left(1 + \frac{X}{N}\right)^N$$

for $-1 \leqslant X \leqslant 1$ where the integer N determines the accuracy. Take care when applying the scaling to avoid errors due to rounding or overflow. How does the resulting method compare with the version given in the text for speed and accuracy?

REFERENCES

[1] Bumgarner, J.O. 'Fixed-Point Trigonometry by Derivation' *Forth Dimensions*, **IV**, No. 1, p7

[2] ** i.e. X^n is from *Forth Dimensions*, **IV**, No. 3, p31

[3] Grossman, N. Jan/Feb 1984. 'Fixed Point Logarithms' *Forth Dimensions*, **V**, No. 5, p11

7 Calendar Functions

7.1 JULIAN DATE

Astronomers require a convenient calendar that covers long periods of time before and after the present day and which can also allow for the 11 days lost in changing from the Julian calendar to the present Gregorian calendar. The Julian day number is the number of days counted from 1 January 4713 BC since this was originally thought to be roughly when the world began.

The program definition JULIAN leaves the Julian day as a double number, given the date in the new style (Gregorian) calendar.

```
( JULIAN DATE FROM ASTROPHYSICAL J. SUPPL. V41/3 11/79          )

:JULIAN ( DAY MONTH YEAR --- D /LEAVES JULIAN DATE AS DBLE #  *)
        >R DUP 9 +  12 /  R +  7 *  4 /  MINUS
        OVER 9 -  7 /  R +  100 /  1+  3 *  4 /  -  SWAP
        275 9 */  + +  S->D   1.721029 D+  367 R> M*   D+  ;

( E.G.  5 11 1888  JULIAN D. 2410947 OK )

```

MINUS should be replaced by NEGATE if necessary. The routine will cope with dates BC if they are input as negative and will go back to circa 4600 BC. Day one BC is entered as 31 12 −1 and is one day before 1 1 0000 (J = 1 721 060). Beware of placing too much reliance on negative years as discrepancies of a few days have been noted. For example the equation appears to give day one as 25 November −4713 and JULIAN fails at around 1 3 −4682.

The definition of JULIAN uses the formula

```
J = 1721089 + D + INT(367((M-2)/12+x))
    + INT(INT(365.25(Y-x))-0.75k)
```

where

x = 1 if M = 1 or 2
k = 2 for the old style Julian calendar
k = INT((Y-x)/100) for the new style Gregorian calendar.

Both systems give the same Julian day number, but remember that JULIAN as defined here computes the equation based on the Gregorian calendar. And yes, the numbers in JULIAN (1.721 029) and in J (1 721 089) are supposed to be different.

7.2 WHAT DAY IS IT?

Several algorithms have been published to calculate the day of the week from the date for a limited range of years. Most of them appear to be some simplification of an algorithm known as Zeller's Congruence. Screens 137 to 139 are for a program to print a calendar for the given month and year and screens 137 and 150 are for the day of the week.

For the day of the week the date can be entered as the day number, month as the first three characters and the year in full, for any year in the Gregorian calendar system from AD1 to the year 4902. Since the Gregorian calendar was not introduced until 15 October 1582, before this you will need to make a correction. Years less than one are inadmissible.

```
SCR #137
( CALENDAR - 1 OF 3                                   RDG-821022 )

: DATA      <BUILDS , , DOES>  DUP @  SWAP  2+ @  ;
31  1 DATA JAN         28  2 DATA FEB         31  3 DATA MAR
30  4 DATA APR         31  5 DATA MAY         30  6 DATA JUN
31  7 DATA JUL         31  8 DATA AUG         30  9 DATA SEP
31 10 DATA OCT         30 11 DATA NOV         31 12 DATA DEC

: ->DAYS  ( MO YR --- D /CONVERT TO DAYS SINCE JAN OF YR 0    *)
        OVER OVER 365 M* ROT 1- 31 M* D+ >R >R SWAP DUP 3 <
        IF  DROP 1-  0  ELSE  4 *  23 +  10 / MINUS  THEN
        SWAP DUP  4 /  SWAP 100 /
        1+   3 4 */  - +  0  R> R>  D+  ;



SCR #150
( DAY OF THE WEEK                                     RDG-821220 )

: D0   ." SATURDAY"  ;            : D1   ." SUNDAY"  ;
: D2   ." MONDAY"  ;              : D3   ." TUESDAY"  ;
: D4   ." WEDNESDAY"  ;           : D5   ." THURSDAY"  ;
: D6   ." FRIDAY"  ;

: .DAYN ( --- /PFA IS A LIST OF CFAS TO PRINT DAY                *)
        D0 D1 D2 D3 D4 D5 D6  ;

: DAY?  ( N --- /PRINT DAY FROM DAY NUMBER                       *)
```

```
          ' .DAYN  SWAP 2 * + @ EXECUTE  ;

 : DAYOWEEK  ( DD <mth> YYYY --- /PRINT DAY OF WEEK FROM DATA  *)
         SWAP DROP  ->DAYS  ROT  0 D+  7 M/MOD DROP DROP
         ." IS A " DAY?  CR  ;
```

Here is an example of its use:

```
1 JAN 1985 DAYOWEEK IS A TUESDAY
 OK
```

The program uses a variation of Zeller's congruence to calculate the number of days elapsed since year dot. Zeller's expression involves some clever footwork to account for leap years and their occasional absence at the turn of certain, but not all centuries. To keep the arithmetic simple, two expressions are used:

```
Days = 365(YYYY) + DD + 31(MM-1) - INT(0.4*MM + 2.3) + INT(YYYY/4)
         - INT(3/4(INT(YYYY/100)+1))
```

for March to December or

```
Days = 365(YYYY) + DD + 31(MM-1) + INT((YYYY-1)/4) -
         INT(3/4(INT(((YYYY-1)/100) + 1)))
```

for January to February.

These expressions are evaluated as a double-precision number by ->DAYS and M/MOD converts them to a number modulo 7 for the day of the week. The parameter field of .DAYN contains a list of cfas which are addresses of words to print each day of the week. DAY? uses the stack value left by the remainder from M/MOD as an index to the list, fetches the indexed cfa and executes it. Note that the word ' (tick) must return the pfa of .DAYN because it may differ in some dialects of Forth.

To enable the month to be entered as text, a defining word DATA is defined which will create a new family of words whose characteristics are similar to a double-precision constant. When the month is called by name, it returns its month number and last day. The last day is discarded by DAYOWEEK but is used in CALENDAR which follows.

7.3 CALENDAR

The calendar definition is called by name of month and year, for example NOV 1888 CALENDAR will result in the calendar for that month being displayed on the terminal.

```
SCR #138
 ( CALENDAR - 2 OF 3                                      RDG-821022 )

 : HEADER  ( --- /PRINT THE HEADER                                *)
         CR CR ."  SUN MON TUE WED THU FRI SAT"
```

```
        CR SPACE  27 0 DO  45 EMIT  LOOP  ;

: 1STDAY  ( N1 N2 --- N3 /MTH YR TO DAY OF WEEK, 0 = SUNDAY   *)
        ->DAYS 7 M/MOD DROP DROP ;

: WEEK1   ( N --- N+7 /FORMAT 1 WEEK IF N RANGES 1 TO MTH.END *)
        CR 7 0 DO  DUP 1 <
                   IF    4 SPACES
                   ELSE  SPACE DUP 10 <
                         IF  SPACE  THEN  DUP MTH.END @  >
                         IF  4 SPACES  ELSE  DUP .  THEN
                   THEN  1+ LOOP  ;                -->

SCR #139
( CALENDAR - 3 OF 3                                 RDG-821022 )

: ?LEAP ( N --- /MAKE MTH.END = 29 IF N = LEAP YEAR          *)
        DUP  0 4 M/  DROP  0=
        IF    DUP  0 400 M/  DROP  0=
              IF    29 MTH.END !
              ELSE  DUP 0 100 M/ DROP 0= 0=
                    IF   29 MTH.END !  THEN
              THEN
        THEN  ;

: CALENDAR   ( <mth> N --- /CALENDAR FROM 3 CHAR MONTH & YYYY *)
        SWAP  DUP MTH.END !  28 =  IF  ?LEAP  THEN
        HEADER  1STDAY DUP MINUS  1+
        SWAP  MTH.END @ +  6 +  7 /
        0 DO  WEEK1  LOOP  DROP CR CR CR  ;
```

NEGATE should be used instead of MINUS in '79-standard or later. An example format is shown for February 2000 which occurs in a leap year.

```
FEB 2000 CALENDAR

 SUN MON TUE WED THU FRI SAT
 ---------------------------
              1   2   3   4   5
  6   7   8   9  10  11  12
 13  14  15  16  17  18  19
 20  21  22  23  24  25  26
 27  28  29
```

Note that a year is a leap year if it is divisible exactly by 4, or 400, but not if by 100 (or 4000?). This complex correction factor is to make a calendar year equal to 365.2425 days. A year is defined as the time of one earth orbit round the sun and a day to be one rotation of the earth about its axis. The ratio of these two, i.e. the number of days in a year, is 365.2422 ...., an irrational number, the value of which is also complicated by other factors, not to mention the earth's rotation very gradually slowing down!

Project

Investigate the limitations of the JULIAN definition for BC dates. Determine at what future date JULIAN will fail.

8 Factors and Multiples

8.1 FACTORS

Much fun was had playing with the factors routine, for example the highest prime number that the author could find was 32749, whereas 32738 = 2 × 16369 and 30030 is the product of the first six primes.

There are several methods for finding factors, the one here works by trial and error trying each prime in turn. Typical formats are:

```
5600 FACTORS = 1 * 2 ↑5 * 5 ↑2 * 7
5601 FACTORS = 1 * 3 *  1867
```

The leading 1 serves to remind one that it is also a factor, but its real purpose is to avoid the difficulty of suppressing the leading *.

```
( FACTORS - 1 OF 2                                  RDG-840528 )

: PRINT  ( N1 N2 QUOT COUNT --- N1/N2 OR N1  /N1 IF COUNT=0   *)
         -DUP IF  ROT  ." * " .  ( N2 )  DUP   1 >
               IF  ." ↑" . ( Count )  ELSE  DROP  THEN DROP
              ELSE  DROP DROP    ( leave N1 )
              THEN  ;

: TOTHE  ( N1 N2 --- N3 /LEAVES N1/N2 OR N1 IF NOT INTEGRAL   *)
         0 BEGIN  >R OVER OVER  /MOD   R>  ROT
                  ( N1 N2 quot count rem )  0=
           WHILE  1+  >R >R  SWAP DROP  R> SWAP  R>
           REPEAT
         PRINT  ;
-->

```

```
( FACTORS - 2 OF 2                                  RDG-840528 )

: CVECTOR   ( N -- /Compile N bytes off stack                 *)
        <BUILDS  0 DO C, LOOP
        DOES>    + C@  ;

```

```
 6         31 29 23 19 17 13 11 7   8 CVECTOR TRIAL
 7
 8 : FACTORS   ( N --- /PRINTS OUT ANY FACTORS OF N 32767 MAX.   *)
 9         ." = 1 "
10         2 TOTHE  3 TOTHE  5 TOTHE
11         DUP 1 =  IF  CR DROP  EXIT  THEN
12         DUP 0 DO  8 0 DO  I TRIAL J + TOTHE  LOOP
13                   DUP I < IF  LEAVE  THEN  30
14              +LOOP  DROP  ;
15
```

The single dimension byte array TRIAL contains numbers which are not multiples of 2, 3 or 5, i.e. the prime numbers from 7 up to 31. These numbers are used as trial divisors in TOTHE. However, they do not all have to be prime and, in fact, the series is repeated with 30 added each time. This will result in occasional wasted trials, e.g. 30 + 19 = 49 may be tried, even though 7 will have already eliminated it as a factor.

PRINT handles the format of the printed output and adjusts the stack parameters ready for the next trial. TOTHE extracts as many as possible of a given trial and raises the count used by PRINT. CVECTOR creates a self-loading byte array and is used to create TRIAL which returns the byte indexed into the array. FACTORS then tries 2, 3 and 5 before using the trials in TRIAL. By successively adding 30, the numbers in TRIAL are reused until the trial is no longer less than the number to be factored.

8.2 PRIME NUMBER GENERATION

Now that you have a program to factorize numbers, how about some numbers that do not factorize? One of the classical methods of generating prime numbers is by using the Sieve of Eratosthenes [1], [2]. The first four numbers 0, 1, 2, and 3 are known to be primes and this is assumed, together with the fact that all even numbers are not prime, neither are all multiples of primes. The program uses these facts to successively work through an array of flags. At line 8, for each position corresponding to 'twice the current array index plus 3' the content is cleared from true to false. In addition, for each index, it clears other flags that cannot yield a prime. As can be imagined, if you want a lot of prime numbers, the array needs to be quite large. In the program shown a size of 1200 odd bytes yielded 2399 as the highest prime. Obviously if you have more memory at your disposal (and more time for program execution) then a greater number can be generated. Using 8190 for SIZE will yield primes up to 16381. The word PRIME searches through the array and prints out the number corresponding to the array index if its flag is still set.

```
0 ( SIEVE OF ERATOSTHENES - PRIME NUMBER PROGRAM.     RDG-841006 )
1      1200 CONSTANT SIZE    ( 2399 HIGHEST PRIME )
2         0 VARIABLE FLAGS   SIZE ALLOT
3 : SIEVE     ( --- /SET STATE OF FLAGS IN ARRAY                *)
```

```
             FLAGS SIZE 1 FILL  ( set array )   0  ( count = 0)
             SIZE 0 DO  FLAGS I + C@
                        IF   I DUP +  3 +  DUP I +
                             BEGIN  DUP  SIZE <
                             WHILE  0 OVER FLAGS + C!  OVER +
                             REPEAT   DROP DROP 1+
                        THEN
                    LOOP  . ." PRIMES"  ;

 : PRIMES  ( --- /DO PRIME NUMBERS ACCORDING TO ARRAY FLAGS    *)
            CR SIZE 0 DO  FLAGS I + C@
                          IF  I DUP +  3 +  5 .R  THEN  LOOP  ;
```

The action of SIEVE is sufficiently time consuming to form a useful benchmark program [2], although it is wise not to attach too much importance to benchmarks.

8.2.1 Test for prime

Sometimes it suffices to know that a number has no factors, i.e. is prime. Basically the technique puts a true flag on the stack and then looks for factors in *N*. If the search fails the number is prime and the true flag remains. If a factor is found, the flag is dropped and replaced by a false flag and then the search is terminated.

```
: ?PRIME       ( N --- /LEAVE TRUE FLAG IF N IS PRIME            *)
               TRUE  OVER 2 /  1+  3
               DO    OVER  I MOD  0= IF  DROP FALSE LEAVE  THEN
               LOOP  SWAP DROP  ;
```

For '83-standard TRUE should be −1, for other standards +1 is more common. It should be the same as returned by the standard in use in the interests of consistency.

8.3 HIGHEST COMMON FACTOR

Given two numbers, the highest common factor is the greatest number that will divide exactly into both. On the American continent it is called the greatest common divisor (GCD).

```
: HCF          ( N1 N2 --- N3 /N3 IS HCF OF BOTH N1 & N2         *)
               BEGIN  SWAP OVER  MOD  -DUP 0=  UNTIL  ;

: .HCF         HCF CR ." The HCF is " .  ;
```

The HCF presented here performs three operations known as Euclid's algorithm and is one of the earliest known algorithms, dating back as it does to c. 300 BC. The steps are:

1. Divide N1 by N2 and leave the remainder.
2. If the remainder is zero leave with N2 as the answer.
3. Replace N1 with N2, N2 with the remainder and go back to 1.

You should have little difficulty relating these steps to the program once you realize that step 3 is performed first! In the example -DUP should be replaced by ?DUP in later standards.

8.4 LOWEST COMMON MULTIPLE

Following naturally from HCF is LCM, and the lowest common multiple is given by:

```
: LCM          ( N1 N2 --- D /Leave lowest common multiple        *)
               >R 0 OVER R SWAP R>  HCF  M*/  ;

: .LCM         ( N1 N2 --- /Print LCM                             *)
               LCM CR ." LCM = " D.   ;
```

The LCM is given simply by N1*N2/HCF, but since N1 and N2 could both be prime, the result could be a double number. The definition above accommodates this possibility.

8.5 FACTORIALS

Factorials occur in many mathematical expressions, such as formulae for permutations and combinations. Factorial *N* grows large very rapidly with increasing *N*.

```
: N!           ( N --- N! /Compute factorial N - [8 max]          *)
               1 SWAP 1+  1 DO   I *   LOOP  ;
```

The maximum permissible number is 8 since 8! = 40320 but this can be increased by using double-precision arithmetic to leave a 32-bit result.

```
: DN!          ( U --- D /Compute factorial N leave double #      *)
               1. ROT 1+  1 DO  I UM*  LOOP  ;
```

The double number version, DN!, will work with *N* up to 12 before the result is too large. Both definitions return 1 for a negative number. UM* requires a double number with a single number on top and leaves a double number product.

Factorials are an obvious candidate for recursion, or so we are told, but in Forth we can actually lose out due to the necessary stack manipulations. Recursive versions of the above require the previously mentioned word MYSELF which compiles the current definition's compilation address into itself. In addition the double number version requires the version of UM* which multiplies a double number on the stack by a single-precision number on top. Line 9 corresponds to UM* and may be factored out if UM* is already defined.

```
SCR #136
 0 (FACTORIAL-RECURSIVE VERSION                            RDG-841019 )
 1
 2 : MYSELF    LATEST  PFA  CFA ,  ; IMMEDIATE
 3
 4 : FACT      ( U1 --- U2 /RETURN FACTORIAL OF TOS VALUE         *)
 5             DUP 1 >  IF    DUP  1-  MYSELF  U* DROP  THEN  ;
 6
 7 : DFACT     ( U --- D /GIVES FACTORIAL OF TOS AS DOUBLE #      *)
 8             DUP 1 >  IF    DUP  1-  MYSELF  ROT
 9                            DUP ROT * -ROT U* ROT +
10                      ELSE  0  THEN  ;
11
12
13
14
15
```

The recursive version puts on the stack N, $N-1, \ldots 3, 2, 1$ by successively repeating DUP 1- until failing the 'greater than one' test. The ELSE branch converts the 1 on top to double-precision by capping it with a zero. The definition now unwinds by resuming with the words following MYSELF. ROT moves in turn each stack item over the double number, where line 9 (UM*) leaves a double product ready for the next item. Finally there is nothing left but the double-precision result. The limiting values are as before, but negative values behave differently. If U is negative, DFACT returns U 0, i.e. U converted to double-precision.

Project

Redefine FACTORS ?PRIME and HCF to handle double-precision numbers.

REFERENCES

[1] Sept. 1981. *BYTE Magazine*, p180

[2] Gilbreath, J. and Gilbreath, G. Jan. 1983, *BYTE Magazine*, p283

9 When 10 Digits Are Not Enough

The absence of floating point arithmetic is normally no disadvantage when Forth is appropriate to the task in hand. Even so, the ease with which Forth lends itself to many applications does sometimes result in an apparent requirement for floating point operation. In fact what may be required is simply extended arithmetic precision for certain calculations. The technique is to perform the calculation as you would long hand, but using Forth to do the spadework. For this Forth will need an accumulator or buffer to use for intermediate calculations and for the final result.

9.1 LARGE FACTORIALS

We saw in Chapter 8 how quickly factorials outstripped single- and double-precision calculations. By defining an accumulator as long as is required, we can store product terms without limit (memory permitting). In the example an accumulator of 200 digits enables factorials up to 120! to be calculated. On an average Forth system, the 158 digits of 100! took nearly 50 seconds to calculate and display.

```
SCR #140
( LARGE FACTORIAL - 1 OF 3                              RDG-841227 )

: C.ARRAY   <BUILDS  ALLOT
            DOES>  +  ;

  200 CONSTANT  SIZE   (  120! MAX )
 SIZE  C.ARRAY  !BUFF
    0 VARIABLE  LAST

-->
```

```
SCR #141
 0 ( LARGE FACTORIAL - 2 OF 3                              RDG-841227 )
 1
 2 : *N        ( N --- /COMPUTE NEW PRODUCT IN !BUFF                *)
 3             0  ( initial carry ) LAST @  1+ 0
 4             DO   OVER I !BUFF  C@ * + ( times N, add carry )
 5                  10 /MOD  SWAP I !BUFF C!  ( new carry )
 6             LOOP
 7             BEGIN  -DUP     ( handle final carry )
 8             WHILE  10 /MOD  SWAP 1 LAST +!  LAST @ DUP 1+
 9                    SIZE > IF  ." Overflow" QUIT  THEN
10                    !BUFF C!
11             REPEAT  DROP  ;
12
13 : SETUP     ( --- /INITIALIZE FACTORIAL BUFFER AND POINTER       *)
14             1 0 !BUFF C!  ( SET BUFFER TO 1 )
15             0 LAST !   ;        -->

SCR #142
 0 ( LARGE FACTORIAL - 3 OF 3                              RDG-841227 )
 1
 2 : .FACT     ( --- /SHOW FACTORIAL IN !BUFF WITH COMMA PER K      *)
 3             LAST @ 1+ 0
 4             DO    LAST @ I -  DUP 1+  3 MOD 0=
 5                   I 0=  0= AND IF  44 EMIT ( comma )  THEN
 6                   !BUFF C@  1 .R
 7             LOOP  CR  ;
 8
 9 : FACT      ( N --- /COMPUTE FACTORIAL IN !BUFF                  *)
10             SETUP  1+ 1 DO  I *N  LOOP  ;
11
12 : N.FACTS   ( N --- /DISPLAY ALL FACTORIALS UP TO N!             *)
13             SETUP 1+ 1
14             DO  I *N  CR ." FACTORIAL"
15                 I 3 .R  ." = " .FACT   LOOP  CR  ;
```

Screen 140 defines a word to create a byte array and this is done in line 6 where the factorial buffer !BUFF is created. <BUILDS is a FigForth word and should be replaced by CREATE for '79 and later standards. The variable LAST points to the last product term in the buffer. The word *N on screen 141 does most of the work. It multiplies the contents of !BUFF by the number on the stack. This number should not exceed about 3200 (3200! requires a lot of memory) because the carry on some digits may exceed 32767. A fully implemented multiplication routine should be used to overcome this limitation. *N uses a simple multiply and add sequence. The carry comes from the division by 10, the result of which may leave a carry greater than 10. The final carry is handled by the BEGIN/WHILE/REPEAT loop which extends the buffer by

as many digits as required. If this results in exceeding the length limit of the buffer, the calculation is abandoned by QUIT following the error message. Otherwise -DUP (or ?DUP) generates a true condition, the buffer is extended until the carry is reduced by division to zero whereupon the loop terminates.

SETUP puts 1 in location 0 of the buffer and LAST is set to index to it. The factorial is calculated by FACT which repeatedly calls *N to accumulate the product terms. At any time .FACT can be called to print the buffer contents. It does so from the high digit down, inserting a comma as required to partition each thousand by the words 3 MOD 0= with I 0= 0= AND preventing a comma occurring at the end.

Finally, N.FACTS will print out all the factorials up to that on the stack. This operation is made simple because .FACT prints out the buffer contents non-destructively.

9.2 MULTIPLICATION

If the numeric operation of multiplication is broken down into its step by step procedure, as if one were doing it by hand, then no number is involved which is too large for integer arithmetic. It follows that if the two numbers are stored as a digit sequence in a work area, then the individual digits may be manipulated at will. As when using a slide rule (remember them?), in the program that follows no attempt has been made to account for negative values, nor to handle decimal points.

```
SCR #86
( 'INFINITE' PRECISION MULTIPLY - 1 OF 4            RDG-841228 )

: C.ARRAY   <BUILDS   ALLOT
            DOES>     +  ;

            20 CONSTANT <LEN>
       <LEN>   C.ARRAY 1BUFF
       <LEN>   C.ARRAY 2BUFF
 <LEN>      1+ C.ARRAY *BUFF ( PART PRODUCT )
 <LEN> <LEN> + C.ARRAY RESULT ( ACCUMULATOR )

: LEN       ( ADDR --- N /LEAVE LENGTH OF STRING AT ADDR      *)
            DUP DUP BEGIN  C@  0= 0=    ( NULL CHAR? )
                    WHILE  1+ DUP       ( NEXT ADDR )
                    REPEAT SWAP -   ( PTR-BASE = LEN )  ;
-->

SCR #87
( 'INFINITE' PRECISION MULTIPLY - 2 OF 4            RDG-841228 )

: ENTRY ( ADDR --- F /GET INPUT TEXT TO ADDR AS BINARY        *)
        PAD  <LEN> 1 - EXPECT   PAD LEN OVER C!  COUNT 0
        DO    PAD I + C@  BASE @ DIGIT
              IF    OVER I + C!   ( STORE DIGIT )
              ELSE  DROP 0 ( LEAVE FF )  LEAVE  THEN
        LOOP   ;

: SUM   ( N --- /ADD  *BUFF * BASE↑N  TO RESULT                *)
        0 ( CARRY )  0 RESULT C@  0 *BUFF C@  MIN   0 ( LIMS )
        DO    OVER I + 1+ REUSLT  DUP C@ ( CONTENT )  ROT
              I 1+ *BUFF C@   ( CONTENT ) + +
              BASE @ /MOD SWAP ROT C! ( LEAVE CARRY )
        LOOP  SWAP  0 *BUFF C@ + 0 RESULT C! -DUP
        IF  1 0 RESULT +! 0 RESULT C@ RESULT C!  THEN  ; -->

SCR #88
( 'INFINITE' PRECISION MULTIPLY - 3 OF 4           RDG-841228 )

: PRODUCT   ( N --- /LEAVE IN *BUFF NUMBER-IN-1BUFF * DIGIT N *)
            0 ( CARRY )  0 0 *BUFF C! ( INIT )  0 1BUFF C@  0
            DO    OVER     0 1BUFF C@  I -   1BUFF C@
                  * +  ( TIMES N, + CARRY )
                  BASE @ /MOD SWAP  I 1+ *BUFF C!
            LOOP  SWAP  DROP 0 1BUFF C@  0 *BUFF C! -DUP
            IF  1 0 *BUFF +!  0 *BUFF C@ *BUFF C!  THEN  ;

: PROMPT    ( --- /PROMPT FOR TWO NUMBERS                       *)
            BEGIN  CR ." 1ST NUM ? " 0 1BUFF ENTRY  UNTIL
            BEGIN  CR ." 2ND NUM ? " 0 2BUFF ENTRY  UNTIL  ;
-->



SCR #89
( 'INFINITE' PRECISION MULTIPLY - 4 OF 4           RDG-841228 )
: INIT  ( --- /INITIALIZE THE RESULT BUFFER                    *)
        <LEN> <LEN> + 1 -  1 RESULT OVER ERASE  0 RESULT C! ;

: TIMES ( --- /MULTIPLY 1BUFF & 2BUFF CONTENTS
        0 2BUFF C@ DUP 0   DO  DUP I - 2BUFF C@  ( GET MULT )
        -DUP  IF  PRODUCT  I SUM  THEN  LOOP DROP  ;

: .ANS  ( ADDR --- /GET COUNT AT ADDR & PRINT DIGITS            *)
        COUNT SWAP OVER 0  CR
        DO   OVER I - 3 MOD 0=  I 0= 0= AND  IF 44 EMIT THEN
             DUP 1 - C@  I -  OVER +  1 -  C@  1 .R
        LOOP DROP DROP  ;

: MULT ( --- /INPUT TWO NUMBERS, LEAVE RESULT IN BUFFER         *)
       INIT  PROMPT  TIMES  0 RESULT .ANS  ;
```

C.ARRAY is a defining word to create and determine the action of the work areas required. At compile time it allots the specified amount of memory. At run time the stack value is used as an index to the buffer by adding its value to the base address.

LEN is used by ENTRY to determine the length of the input string. It works by examining each character from the first until a zero is found.

ENTRY accepts characters into PAD and converts from ASCII to numbers in the current base using DIGIT. Conversion stops at zero or the first invalid character. A false flag is left if the conversion fails so that PROMPT can request a retry. Note that in Forth-'83 standard, EXPECT does not put a zero (required by LEN) at the end of the string, but leaves the character count in SPAN.

PRODUCT multiplies the number 1BUFF by the given digit and leaves the result in *BUFF. Since numbers are entered Most Significant Digit (MSD) first, and we require the LSDs aligned, the result is written to *BUFF Least Significant Digit first.

SUM accumulates the contents of *BUFF in RESULT. The value of *BUFF can be offset by the stack value *N* to allow for the numeric significance of the number in *BUFF, i.e. the number is multiplied by the current base to the power *N*.

INIT initializes only the result buffer, the others are simply overwritten and the count updated accordingly. TIMES does the real work of multiplying the two numbers and .ANS simply prints the answer. MULT does the whole thing, prompting for input and displaying the answer.

9.3 DIVISION

The same technique used for multiplication can be applied to division. In this case marking the decimal place is of more value. The basis of the routine was published in *COMPUTE!* magazine [1] but was revised to output to a work-space called WKSPC in addition to the terminal device.

```
SCR #125
 0 ( O/P AS FLPT, DIVISION OF TWO NUMBERS - 1 OF 2     RDG-840928 )
 1                                                ( AFTER MG-AUG 1983 )
 2
 3   0 VARIABLE WKSPC 15 ALLOT   ( DEFINE A WORKSPACE )
 4
 5 : FRESH      ( --- /FILL BUFFER WITH SPACES                  *)
 6              WKSPC 16 OVER C!  ( PUT COUNT IN 1ST POSITION )
 7              COUNT  ERASE  ;
 8
 9 : C.EMIT     ( C --- /OUTPUT CHARACTER TO WKSPC & TERMINAL    *)
10              DUP EMIT  WKSPC DUP >R C@ 1+  DUP R C! R> + C!  ;
11
12 : 1TO3       ( N1 N2 N3 --- N1 N3 N2 N3 /DUP TOS INTO 3RD POSN *)
13              DUP  ROT  SWAP  ;
14  -->
15
```

```
SCR #126
( O/P AS FLPT, DIVISION OF TWO NUMBERS - 2 of 2    RDG-840928 )

: QUOT      ( N1 N2 --- /COMPUTE & OUTPUT N1/N2 AS FLTPT      *)
            1TO3 /MOD   0 <# #S #>  DUP WKSPC !  2DUP TYPE
            WKSPC 1+ SWAP  CMOVE 10 * SWAP 1TO3  46 C.EMIT  ;

: REMAIN    ( N1 N2 --- /OUTPUT NEXT DIGIT OF REMAINDER       *)
            /MOD 48 + C.EMIT   10 * SWAP 1TO3  ;

: FP/.      ( NUM DENOM --- /OUTPUT 10 DIGITS OF REMAINDER    *)
            DECIMAL  FRESH  QUOT
            10 0 DO  REMAIN  LOOP DROP DROP DROP  ;
```

WKSPC is a buffer created by a FigForth definition. Other standards may use CREATE WKSPC 17 ALLOT as an equivalent.

FRESH clears the buffer ready for use. C.EMIT outputs the character on the stack to the terminal and additionally stores it in WKSPC. 1TO3 is a factored definition to rearrange the stack contents. QUOT handles the whole number portion of the result and REMAIN is called for each digit of the fractional part. FP/. does the work of getting it all together.

The three routines given for multiplication, division and factorials have come together in this chapter from three different thought directions. As a result they do not form a neat set of definitions, they are not very good Forth and they have plenty of room for improvement. They have been left in the raw state in order to illustrate what improvements should be made and to save the author from having to rewrite them all! Now for the critique:

1. Each routine has its own dedicated sets of working space, when in practice they could share that of, for example, the multiplication routine. It follows that the three different initialization words could be commoned and *N and PRODUCT become the same thing and .FACT and .ANS are essentially the same.
2. The multiplication and division routines keep a count of the numeric string length, whereas the factorial uses pointers.
3. Multiplication uses the numeric equivalent of each digit, whereas division uses ASCII characters.
4. The use of dedicated buffers prevents the routines from being used to process numbers in other parts of a system without having to move them first.
5. In the multiplication routine, TIMES does not perform the product and sum if the digit in question is zero, but if the digit is one, the product is superfluous and only the sum need be performed.

6. The multiplication routine will work with any reasonable base and this facility could readily be incorporated in the other two. In which case one should consider factoring out `BASE @ /MOD SWAP` as a separate definition.
7. The choice of names could be improved as part of the tidying up process.

The above points illustrate a Forth programming rule: 'write the general, not the specific'. Each routine described came from a different application and was specifically tailored to that application, whereas had they been written in a more general form they would not have come together as such complete strangers.

Project

Respond to the above critique. From the examples in this chapter create a revised set of definitions to handle 'unlimited' precision arithmetic in the current base.

REFERENCES

[1] Ganis, M. August 1983. 'Floating Point Division' *COMPUTE!* p249

10 Pot-pourri

10.1 SORT AND SEARCH

The subjects of search and sorting are sufficiently large to warrant a text of their own [1]. Few routines of this nature have appeared written in Forth. Numbers may be sorted recursively on the stack [2], but this precludes using the stack for anything else during the sort. The standard quicksort routine has been implemented in Forth [3] and it is included here primarily because the search routine chosen requires a sorted list! One search and one sort have been selected by way of example.

10.1.1 Quicksort routine

The quicksort technique is to divide the array into two and compare the middle value with each item in the lower end of the array until a value greater than the mid-value is found. The high end of the array is then searched from the end downwards for a value less than the mid-value. These two numbers are then swapped. The process continues until the middle is reached. At this point we have an array of values less than the mid-value and another of values greater than the mid-value. The process is repeated on the two arrays and again on the four arrays, continuing until an array fragment is so small it is only one item long. With only one item it no longer requires sorting. The process is complete when all pieces have been sorted.

```
( QUICK SORT - SORTS BYTE ARRAY - 4TH DIM V/5 P29    MP-840100 )

0 VARIABLE MIDDLE
: EXCHANGE   ( ADDR1 ADDR2 --- /SWAP BYTES AT ADDR1 & ADDR2   *)
          2DUP C@ SWAP C@ ROT C! SWAP C!  ;

: SORT         ( START.ADDR END.ADDR --- /SORTS BYTE ARRAY      *)
          2DUP 2DUP  OVER - 2/ + C@ MIDDLE ! ( pick middle one )
          BEGIN  SWAP BEGIN  DUP C@ MIDDLE @ < WHILE 1+ REPEAT
                 SWAP BEGIN  DUP C@ MIDDLE @ > WHILE 1- REPEAT
                 2DUP > 0= IF  2DUP EXCHANGE 1 -1 D+  THEN
                 2DUP >                   ( until partitions cross )
          UNTIL  SWAP ROT                        ( sort both pieces )
          2OVER 2OVER -  -ROT  - < IF  2SWAP  THEN
```

```
 14          2DUP < IF  MYSELF  ELSE  2DROP  THEN
 15          2DUP < IF  MYSELF  ELSE  2DROP  THEN  ;
```

EXCHANGE is similar to VSWAP but operates on bytes, not 16-bit values. Lines 7 to 11 perform one dividing pass and leave four addresses on the stack representing the ends of the two smaller arrays. Lines 12 and 13 arrange that the smaller of the two arrays is sorted first to prevent the recursion from going too deeply. Finally, lines 14 and 15 of the sort repeat the process on both arrays by recalling SORT until done. The action can be seen pictorially on computers using memory-mapped video.

10.1.2 Binary search

The binary search is similar in operation to the sort and is a fast way of looking for a given value in a sorted array. Again the middle element is selected. If this is larger than the desired value, then the upper half of the array is ignored, otherwise the lower half is ignored by simply adjusting the array pointers. As can be guessed the halving process is continued until either the required value is found, or the pointers cross paths. In the latter case the search has failed. Note that to search a 4000 element array any item can be found or determined to be absent by examining only 12 elements. An element can be a byte, cell or 32-bit double-precision value without affecting the number of operations.

```
SCR #79
 0 ( BINARY SEARCH ROUTINE                                  RDG-841230 )
 1   0 VARIABLE LOWER          0 VARIABLE UPPER
 2   0 CONSTANT FALSE       0. = CONSTANT TRUE
 3
 4 : SEEK  ( V L U --- N F /SEARCH FOR V BTWN LOWER & UPPER ADDR *)
 5          UPPER !  LOWER !
 6          BEGIN  LOWER @  UPPER @  2DUP >  DUP >R
 7                 IF    DROP
 8                 ELSE  0 SWAP 0  D+  2 U/  SWAP DROP
 9                       2DUP  C@  2DUP =
10                 THEN  R>  SWAP OVER  OR 0=
11          WHILE  DROP  > IF    1+ LOWER !
12                         ELSE  1- UPPER !  THEN
13          REPEAT IF    FALSE
14                 ELSE  2DROP SWAP DROP TRUE  THEN  ;
15
```

SEEK requires on the stack the value sought, the lower address and on top the upper address. The variables LOWER and UPPER are pointers to the array and are initially set to the end points of the array. Line 8 calculates the mid-point, and 9 tests the byte for equality with the sought value. Lines 11 and 12 move the end pointers to their new positions and if or when found, lines 13 and 14 tidy the stack and leave a condition flag. TRUE is defined to leave a true value appropriate to the system in use. In other words this definition is portable to any Forth standard!

10.2 EGYPTIAN AND RATIONAL FRACTIONS

10.2.1 Egyptian fractions

An Egyptian fraction is a fraction which has a numerator of 1 and is so named because the ancient Egyptians lacked practical methods for handling other types of fractions. A fraction whose value is less than 1 is known as a proper fraction and EFRAC is a definition which will partition a proper fraction into a sum of Egyptian fractions. It uses the Fibonacci maximal algorithm published in the year 1202 to do this.

Suppose our fraction is *A*/*B*, we first need the largest Egyptian fraction which is less than *A*/*B*. The difference is taken and the process continued until the difference is zero. In practice a small number can give a fraction whose denominator will exceed 32767 and we then terminate the series with an overflow message. It may be inconvenient in Forth to handle a ratio such as 3/7 but using EFRAC we can see

```
3/7 = 1/3 + 1/11 + 1/231
```

EFRAC is rather complex to explain in words and reference should be made to the flowchart shown in Figure 10.1.

```
SCR #143
( EGYPTIAN FRACTIONS                                  RDG-841228 )

: EFRAC ( N1 N2 --- /PRINT THE EGYPTIAN FRACS = N1/N2          *)
        OVER OVER <  IF   ." = "
        BEGIN  OVER 0=  IF  DROP DROP EXIT  THEN  OVER 1 >
            IF  OVER OVER  SWAP /
                BEGIN   >R  OVER R *  OVER -  R>  OVER 0<
                WHILE   SWAP DROP  1+
                REPEAT  >R -ROT  SWAP DROP  R  U*  R>
            ELSE  SWAP  DROP 0  SWAP  OVER  ( 0 )  OVER ( 0 )
            THEN   ." + 1 / " U.
        UNTIL  ." + OVERFLOW" CR
        THEN  DROP DROP  ;



```

10.2.2 Rational fractions

The handling of rational fractions in Forth has been comprehensively covered by Grossman [4] and will not be discussed in detail here. Rational fractions are useful in Forth's integer arithmetic for handling numbers such as π or the

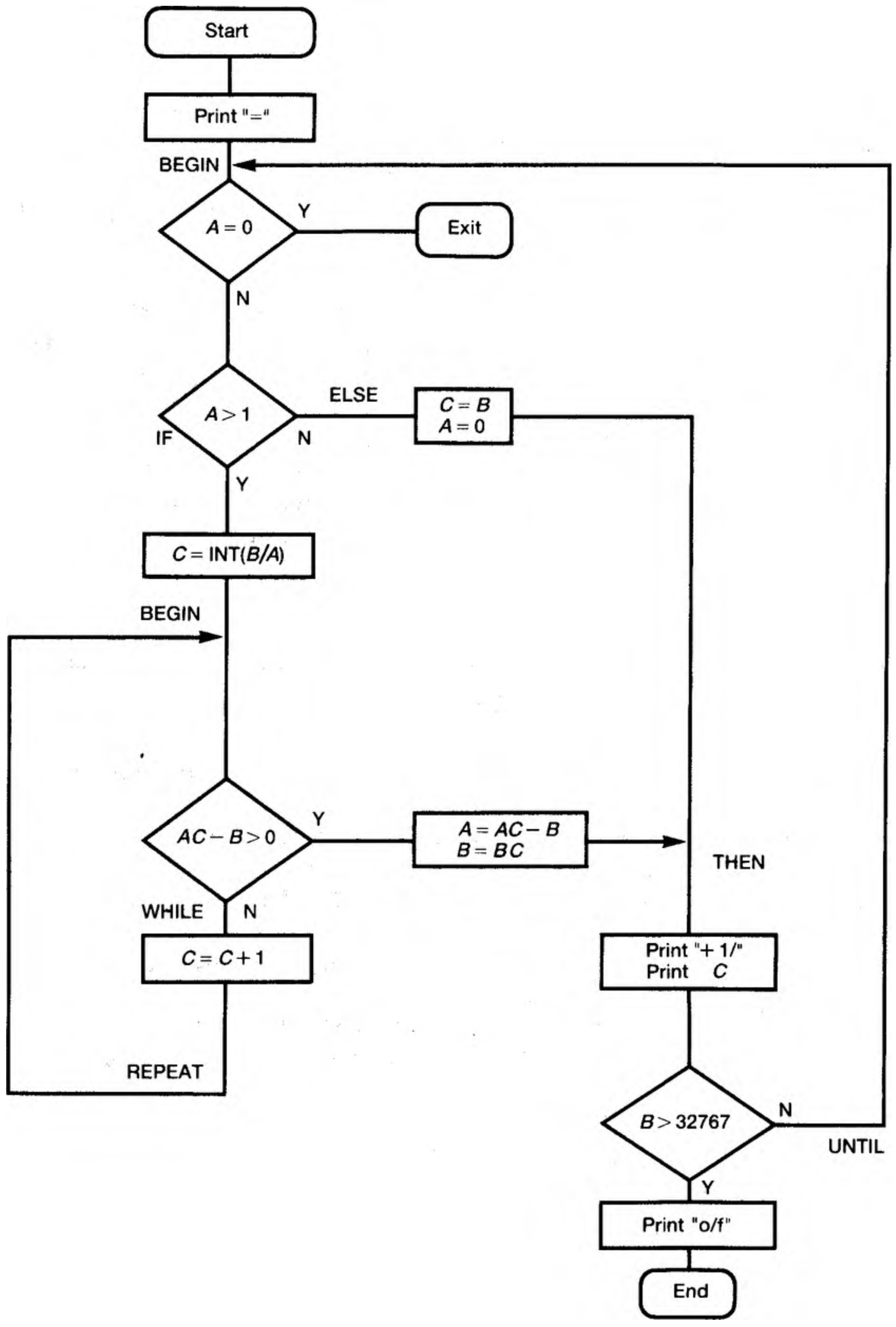


Figure 10.1 Flowchart for EFRAC.

square root of two. The ratios are sufficiently accurate for single-precision, and are generally adequate for double-precision. Some examples are given below.

$$\frac{180}{\pi} = 57.295\,779\,573\,078 = \frac{4068}{71}$$

$$\sqrt{2} = 1.414\,213\,562\,373 = \frac{29\,113}{20\,586}$$

$$\sqrt{3} = 1.732\,050\,807\,5689 = \frac{8733}{5042}$$

10.3 MATRICES AND ARRAYS

Matrices and arrays come in many shapes and sizes and are added to Forth as required. Several examples have already appeared but three FigForth definitions are given here to illustrate the principles. In later standards <BUILDS should be replaced with CREATE. The first is a defining word which at compile time creates a header. At run time the DOES> portion fetches the 16-bit contents of the array element addressed by the index on the stack. The 2*+ was defined as code in Chapter 2 but 2 * + will suffice if speed is not imperative.

```
: TABLE     ( --- /Defines a look-up table of 16-bit elements     *)
            ( N1 --- N2 /Return contents N2 of index N1 )
            <BUILDS
            DOES>  SWAP  2*+  @  ;
```

An example of usage might be as a look-up table where there may not be a simple algorithm to relate the values. Alternatively, for speed, it may be preferred in complex calculations, such as sines and cosines. Rather than type out the whole sine table the problem can be illustrated to perform cubes:

```
TABLE CUBED   0 , 1 , 8 , 27 , 81 , 125 , 216 , (as required)

CR 3 CUBED
 27 OK
```

CMATRIX defines an array *M* by *N*, each element being a single byte. At run time the address is returned of the element m,n which can be used with C@ or C! to fetch or store a single byte.

```
: CMATRIX  ( M N --- /Matrix of 1-byte elements                    *)
           ( M N --- addr /Return address of element m,n)
           <BUILDS  1+ SWAP 1+  DUP ,     * ALLOT
           DOES>    DUP 2+ >R  @  * +      R>  +  ;
```

An example might be for a noughts and crosses game where one defines:

```
2 2 CMATRIX 3X3
```

Note that the addressing of rows and columns goes from 0 to 2 for a 3 by 3 matrix. Consecutive addresses are given by

```
0 0 3X3   1 0 3X3   2 0 3X3   0 1 3X3   1 1 3X3 ...     2 2 3X3
```

MATRIX is similar to CMATRIX but being twice the size returns alternate addresses to handle 16-bit cells. A definition such as this would be used for a 3 by 3 matrix for use in graphics manipulations.

```
: MATRIX   ( M N --- /Matrix of 2-byte elements               *)
           ( M N --- addr /Return address of element m,n)
           <BUILDS  1+ SWAP 1+  DUP ,  2* * ALLOT
           DOES>    DUP 2+ >R  @  * +  2*   R>  +  ;
```

10.4 TOOLS

It is axiomatic that several special purpose tools are better than one general purpose item. In Forth this is especially so since each or all can reside in memory and may be called upon as required. In addition, being specialist they are often, but not always, simpler routines and can be written as required. Some simple examples are included here, but large applications require complex tools and these are best obtained via vendors.

The first special tool here is PLIST which prints two consecutive screens side by side using a 163-column 12-pitch printer and thus allows six screens to be printed per page of 15+ inch listing paper. The words PRT-ON and PRT-OFF must have previously been defined to set and respectively restore printer output.

```
( PLIST - PRINTS 2 SEQ. SCREENS SIDE BY SIDE       RDG-830221 )

: PLIST ( SCR# --- /PRINT THIS SCREEN & NEXT ONE              *)
        PRT-ON CR DUP SCR !     ." SCR #" 4 .R
        71 SPACES     SCR @ 1+ ." SCR #" 4 .R
        16 0 DO     CR I 3 .R SPACE I SCR @    (LINE) TYPE
                       12 SPACES
                       I 3 .R SPACE I SCR @ 1+ (LINE) TYPE
                       ?TERMINAL IF  LEAVE  THEN
             LOOP   CR CR  PRT-OFF ;
```

The word (LINE) takes the line number and screen and converts it to an address and count ready for TYPE to output the line. The number of spaces (71 and 12) may need fine tuning according to the exact number of columns available on the printer.

10.4.1 Cross-reference

It is a useful practice to terminate each line containing a colon definition with a comment marked with an asterisk in a consistent manner. By so doing it enables a simple utility to search the disk and print out all such definitions along with their block and line numbers. The cross-reference utility is reproduced below.

```
 0 ( CROSS-REFERENCE, SEARCH FOR STAR-MARKED COMMENTS  RDG-830519 )
 1
 2 : CROSSREF  ( N1 N2 --- / ...........SEARCH SCREEN RANGE FOR  *)
 3   1+ SWAP   DO  16 0
 4                 DO  FORTH I J (LINE)  C/L =
 5                     IF   62 + @  10538 =
 6                          IF  CR J 4 .R  I 4 .R  I J .LINE THEN
 7                     ELSE DROP
 8                     THEN
 9                 LOOP
10             LOOP  CR  ;
11
12
13         \ LINE MUST END WITH *) IN COLUMNS 63,64 OF LINE.
14
15
```

The 62 + modifies the address left by (LINE) from the first to the 63rd column in the buffer and if the contents equals 10538 then *) is present at the end of line. Note that 10538 = 42 + 256 × 41 where 42 = ASCII * and 41 = ASCII). The comment on line 13 must not be inside brackets since a closing bracket is required within the comment.

10.4.2 Address information

INFO gives details of the address parameters of a colon definition formatted as hexadecimal. Used in conjunction with DUMP it is useful for checking that compilation is as expected. INFO expects the name of a definition to follow it. It came originally, the author believes, from Paul Bartoldi, around 1981.

```
 0 ( INFO - GIVES NFA, LFA, CFA PFA & CONTENTS        RDG-830519 )
 1
 2 : ####  ( N --- /OUTPUTS TOS AS $XXXX                        *)
 3         S->D <# # # # # 36 HOLD #> TYPE 2 SPACES ;
 4
 5 : INFO  -FIND  CR HEX         ( GIVE PARAMETER DETAILS        *)
 6         IF  DROP    DUP NFA ." NFA="    ####
 7                     DUP LFA ." LFA="    ####
 8                     DUP CFA ." CFA="    ####
 9                     DUP     ." PFA="    ####   CR
10
```

```
                     DUP NFA ."   @:" @  ####
                     DUP LFA ."   @:" @  ####
                     DUP CFA ."   @:" @  ####
                             ."   @:" @  ####   CR
        ELSE  ." CAN'T FIND IT ! " CR        THEN DECIMAL  ;
```

A typical format for the word #### might be:

```
INFO ####
NFA=$3197  LFA=$319C  CFA=$319E  PFA=$31A0
  @:$2384    @:$3174    @:$0B87    @:$166F
 OK
```

-FIND is FigForth and returns the parameter field address. FIND is almost the same but returns the code field address. In ths instance some minor adjustments (such as inserting 2+) will be required before use.

Further examples of software tools can be found in references [5] and [6].

Project

Write a short decompiler which will display the cfa in hexadecimal and the corresponding name for each word compiled into the parameter field of a colon definition. Use the form SHOW <name> for the definition and incorporate a check to ensure the word <name> to be decompiled is a colon definition and not some other form. Investigate the problems of extending the idea to cover other types of definition, e.g. variables and constants.

REFERENCES

[1] Knuth, D.E. 1973. *The Art of Computer Programming*, Vol. 3, 'Searching and Sorting'. Addison-Wesley.
[2] Turpin, R.H. July/August 1983. 'Recursive Sort on the Stack' *Forth Dimensions*, V, No. 2
[3] Perkel, M. January/February 1984. 'Quick Sort in Forth' *Forth Dimensions*, V, No. 5.
[4] Grossman, N. 'Long Divisors and Short Fractions' *Forth Dimensions*, VI, No. 3, p10
[5] Anderson, A. and Tracy, M. 1984. *Mastering Forth*. Bowie: Robert J. Bradie Co.
[6] Feierbach, G. and Thomas, P. *Forth Tools and Applications*.

Glossary

The glossary is an alphabetical list of the words defined in this book.

!BUFF	A buffer used for storing the result of the factorial generated by FACT.
####	(n ---) Outputs n formatted as $XXXX
##.	Converts two digits as based 60. used by .HMS
#6	A formatting word in ##. used by .HMS
$XX	(--- n) Leaves the hex value of the characters XX without changing the current base.
$XXXX	(--- n) Leaves the hex value of the characters XXXX without changing the current base.
&X	(--- n) Leaves the ASCII value of the character X.
(1+X).5	A particular form of the binomial series expansion.
(ATN)	A variable used by ATN to accumulate terms in a series.
(COS)	(n1 --- n2) Returns the cosine (scaled by 10000) of n1 where n1 is in radians in the range $\pm\pi/2$ (also scaled by 10000).
(CUBE)	(d n1 --- d n2) Leaves a second approximation n2 from a first guess n1 of the cube root of double number d.
(SIN)	(n1 --- n2) Leaves sine (scaled by 10000) of given angle in radians (also scaled by 10000) for 0 to $\pm\pi/2$.
**	(n1 n2 --- n3) Leaves n1 raised to power n2.
*BUFF	A buffer used by MULT to store partial products.
*N	(n ---) Multiplies content of !BUFF by n. Used by FACT
+MINS	(u1 u2 --- d) Converts u1 degrees and u2 minutes of arc to a double number representing degrees times 65536.
->>	When loading causes loading of next but one screen.
->DAYS	(month year --- d) Returns number of days since January of year 0.
-@	(addr --- n) Leaves inverted contents of address.
-ROT	(n1 n2 n3 --- n3 n1 n2) Rotates top three stack items counter-clockwise.
.ANS	(addr ---) Used by MULT to print out the contents of the RESULT buffer.
.DAYN	Compiles a list of cfas to print the day of the week.
.FACT	(---) Outputs the factorial in !BUFF formatted with a comma separator every thousands digit.

.HCF	(n1 n2 ---) Prints the highest common factor of n1 and n2.
.HMS	(---) Prints the time as HH.MM.SS
.LCM	(n1 n2 ---) Prints lowest common multiple of n1 and n2.
.S	Prints the stack contents non-destructively.
/2↑	(u1 u2 --- u3) Returns u1 right shifted u2 times.
0<>	(n --- f) Returns true flag if n is non-zero.
1+DX	A factored definition to evaluate the next term in a rational polynomial.
1+SX	Similar to 1+DX but for the final term of the series.
1/(1+X).5	A particular form of the binomial series expansion.
10K	A constant defined = 10000
1BUFF	First number input buffer used by MULT.
1STDAY	(n1 n2 --- n3) Converts month number n1 and year n2 to day number of the week for CALENDAR.
1TO3	(n1 n2 n3 --- n1 n3 n2 n3) Used by FP/. to rearrange the stack.
1VIA	An address defined as a constant.
2!	(d addr ---) Stores double number at address.
2*	(u --- u*2) Can be defined to double top stack item or perform arithmetic shift left (unsigned times two).
2*+	(addr n1 --- n2) A code definition which leaves address + 2*n for use with arrays.
2/	(u --- u/2) A code definition to perform an arithmetic shift right.
2↑	(u --- 2↑u) Returns the bit significance of the bit number, i.e. 2^u (range 0 to 15).
2@	(addr --- d) Fetches double number from address.
2BUFF	Second number input buffer used by MULT.
2CON	Defines a 32-bit constant.
2DROP	(d ---) Drops a double number.
2DUP	(d --- d d) Duplicates a double number.
2NUMS	(u ---) Outputs u formatted as two digits.
2OVER	(d1 d2 --- d1 d2 d1) Copies the second double number to the top of stack.
2ROT	(d1 d2 d3 --- d2 d3 d1) Does rotation with three double numbers.
2SWAP	(d1 d2 --- d2 d1) Swaps top two double numbers.
2VAR	Defines a double-precision variable.
2VIA	An address defined as a constant.
32@	(addr1 addr2 --- n) Polls addresses 1 and 2 until a bit is set. Returns n as the bit number set in the range 0 to 15.
3DUP	(n1 n2 n3 --- n1 n2 n3 n1 n2 n3) Duplicates the top three stack items.
3NUMS	(u ---) Outputs u formatted as three digits.
3PICK	(n1 n2 n3 --- n1 n2 n3 n1) Copies third stack item to top. A substitute for 3 PICK.
4#	A format conversion word used by BITS.
4PICK	(n1 n2 n3 n4 --- n1 n2 n3 n4 n1) Copies fourth stack item to top.

	A substitute for 4 PICK.
4SWAP	(n1 n2 n3 n4 --- n4 n3 n2 n1) Transposes top four stack items.
:CODE	Words used in pseudo assembler.
;NEXT	
;POP	
<LEN>	A constant used by MULT.
<MOD>	(n1 n2 --- n3) Limits n1 to the range 1 to n2 inclusive.
?1BIT	(n --- f) Returns a flag which is true if one, and only one bit of n is set.
?E-W	(f ---) Outputs W if true, otherwise E.
?LEAP	(n ---) Checks for leap year and used by CALENDAR.
?N-S	(f ---) Outputs S if true, otherwise N.
?PRIME	(n ---) Leaves a true flag if n is prime.
A-B/A+B	(n1 n2 --- n3) Leaves the ratio of the difference and the sum of two numbers, scaled by 10000.
ACR	Adds a fixed offset to a 6522 VIA chip base address to leave the Auxiliary Control Register address.
APPROX	(n1 n2 --- n1 n3) Computes the next approximation n3 from the first guess n2 for the square root of n1.
ARCCOS	(n --- d) Returns the angle in degrees × 65 536 format whose cosine is the given value (also scaled by 10000).
ARCSIN	(n1 --- n2) Returns the angle in radians (scaled by 10.000) whose sine is the given value (also scaled by 10000).
AREA	(n --- d) Returns the area of a circle from its radius.
ASCII	(<c> --- n) Leaves the ASCII value of the next inline character.
ATN	(n1 --- n2) Returns the angle in radians (scaled by 10000) whose tangent is the given value (also scaled by 10000).
BEARING	Returns the bearing in degrees × 65 536 from the latitudes and longitudes of two points.
BIT-OFF	(c addr ---) Turns off bit c at address.
BIT-ON	(c addr ---) Turns on bit c at address.
BIT-TOGGLE	(c addr ---) Toggles bit c at address.
BIT?	(c addr --- f) Leaves the status of bit c of given address.
BITS	(n ---) Prints n in binary, formatted in blocks of four bits.
BOUNDS	(addr n --- addr+n addr) Returns loop start and end values from start address and count.
C!!	(c ---) Stores a byte in an area which operates as a push-down stack.
C.ARRAY	Defines a byte array.
C.EMIT	(c ---) Used by FP/. to output a character.
CALENDAR	Prints a calendar given the name of the month and year.
CJOIN	(hi lo --- n) Joins two bytes as one cell.
CMATRIX	Defines a matrix of 1-byte elements.
COS	(n1 --- n2) Returns cosine of n1 degrees (scaled by 10000).
CROSSREF	(n1 n2 ---) Searches the range of screens specified and prints lines ending in *).
CSPLIT	(n --- hi lo) Splits the top stack item as high and low bytes.

CSWAP	(n1 --- n2) Interchanges high and low bytes of top stack item.
CTRL-X	(--- n) Leaves the ASCII value for the control character X.
CUBED	A table which returns the value of the index cubed.
CUBEROOT	(ud --- n) Leaves the cube root of the unsigned double-precision number ud.
CVECTOR	A defining word to compile *n* bytes off the stack. At run time returns the byte indexed.
D*	(d1 d2 --- d3) Leaves double-precision product of two double-precision numbers.
D**	(n1 n2 --- d) Leaves n1 raised to power n2 as a double-precision number.
D+-	(d1 n --- d2) Leaves d1 with the sign of n.
D-	(d1 d2 --- d3) Leaves the difference of two double numbers.
D->S	(d --- n) Divides double-precision number (degrees of arc) by 65536 to give single number degrees with rounding.
D/2↑	(ud1 u --- ud2) Returns ud1 right shifted u times.
D0 to D6	Set of words to print the corresponding day of the week.
D0<	(d --- f) Leaves a true flag if the double number is negative.
D0=	(d --- f) Leaves a true flag if the double number is zero.
D2*	(ud1 --- ud2) Performs a 32-bit arithmetic shift left.
D<	(d1 d2 --- f) Leaves a true flag if the top item is greater than the one under it.
D=	(d1 d2 --- f) Leaves a true flag if the double numbers are equal.
D>	(d1 d2 --- f) Leaves a true flag if the top double item is less than the double number under it.
DATA	A defining word to return the month number and number of days in that month for use by DAYOWEEK.
DAY?	(n ---) Prints the day of the week from the day number.
DAYOWEEK	Prints the day of the week given the day, name of the month and full year.
DCOS	(d --- n) Returns cosine (scaled by 10000) of degrees × 65536.
DDRA	Adds a fixed offset to a 6522 Versatile Interface Adapter chip base address to leave the Data Direction Register A address.
DDRB	As DDRA but for the Data Direction Register B.
DEPTH	(--- n) Returns the number of items on the stack.
DFACT	(u --- d) Returns factorial of u as a double-precision number.
DMAX	(d1 d2 --- d3) Leaves maximum of two double numbers.
DMIN	(d1 d2 --- d3) Leaves minimum of two double numbers.
DN!	(n1 --- n2) Leaves factorial of n1.
DSQRT	(d --- n) Leaves the square root of double number d.
DU<	(ud1 ud2 --- f) Leaves a true flag if the top unsigned double number is greater than the unsigned double number under it.
EFRAC	(n1 n2 ---) Outputs n1/n2 as an Eygptian fraction.
ENTRY	(addr --- f) Enters numeric input to address leaving false flag if number in current base is invalid.
EXCHANGE	(addr 1 addr 2 ---) Exchanges the bytes at addresses given.
EXIT	1. Exit immediately from the current definition.

EXIT	2. Stop compilation of current screen.
EXP	(n1 --- n2) Returns the exponential (scaled by 10000) of n1 for the range ±1 (also scaled by 10000).
FACT	(u1 --- u2) Returns the factorial of u1.
FACTORS	(n ---) Prints out the factors of n.
FALSE	(--- ff) A constant representing the false flag.
FLAGS	A table of flags used by SIEVE.
FP/.	(n1 n2 ---) Outputs n1/n2 as a floating point string.
FREE	(---) Outputs the available dictionary space.
FRESH	(---) Initializes the WKSPC buffer used by FP/.
GETDEGS	Arranges stack contents for LAT and LONG.
HCF	(n1 n2 --- n3) Leaves the Highest Common Factor of n1 and n2.
HEADER	Prints the header for a calendar.
HISTORY	(--- addr) Returns the address of the storage area used by C!!
I'	(--- n) Returns DO/LOOP end limit.
IER	Adds a fixed offset to a 6522 VIA chip base address to leave the Interrupt Enable Register address.
IFR	As IER but for the Interrupt Flag Register address.
INFO	(<name> ---) Returns address and contents of the nfa, pfa and cfa of the named word.
INIT	(---) Used by MULT to initialize the RESULT buffer.
J	(--- n) Returns outer loop variable.
JULIAN	(day month year --- d) Leaves the Julian date as a double-precision number.
KN	(n1 n2 --- n3) A common term used in series evaluation for SIN and COS.
LAST	A variable used by FACT.
LAT	(d ---) Displays degrees (times 65 536) formatted as latitude.
LCM	(n1 n2 --- d) Leaves Lowest Common Multiple as double-precision number.
LEN	(addr --- n) Leaves the length of the string at address.
LENGTH	A constant used by C!!
LIMITS	(n1 lo hi --- n2) Limits n1 to within the limits of lo–hi.
LOG2.N	(n1 --- n2) Returns the bit number of the lowest bit set.
LONG	(d ---) Displays degrees (65 536) formatted as longitude.
LOWER	A variable used by the binary search routine SEEK.
M*/	(d1 n1 n2 --- d2) Leaves a double-precision number which is the result of multiplying a double number by the ratio n1/n2.
M+	(d1 n --- d2) Leaves the double-precision sum of a single-precision number and a double-precision number.
M/MOD	(ud1 u1 --- u2 ud2) Divides unsigned double-precision number by unsigned single-precision number leaving an unsigned reminder with a double-precision quotient.
MATRIX	A defining word that creates a table of 2-byte elements.
MIDDLE	A variable used by the quick sort routine SORT.
MINSOUT	Formats minutes of arc for LAT and LONG.
MODULUS	(n1 n2 --- n3) Leaves the square root of the sum of the squares

	of n1 and n2.
MODVAR	A defining word to constrain a number within defined limits.
MULT	(---) Outputs the product of two numbers input from the keyboard.
MYSELF	(---) Compiles the cfa of the current definition.
N!	(n1 --- n2) Returns the factorial of n1.
N.FACTS	(n ---) Computes and displays all factorials up to factorial n.
NDUP	Replicates the top N stack items.
NM2DEG	(n --- d) Converts nautical miles to degrees latitude.
NM>	(d --- n) Converts degrees × 65536 to nautical miles.
ORA	Adds a fixed offset to a 6522 Versatile Interface Chip base address to leave the Output Register A address.
ORB	As ORA, but for the Output Register B.
PCR	As ORA, but for the Peripheral Control Register.
PICK	Picks the *N*th stack item and puts a copy on the top.
PLIST	(scr ---) Prints two screens side by side.
PROMPT	Prompts for two numbers for MULT.
PRIMES	(---) Generates prime numbers.
PRINT	An internal definition used by FACTORS to print the results.
PRODUCT	(n ---) Leaves in *BUFF the value in 1BUFF multiplied by n.
QUOT	(n1 n2 ---) Used by FP/. to output the whole number quotient.
RANGE	Returns the range in nautical miles from the latitudes and longitudes of two points.
REMAIN	(n1 n2 ---) Used by FP/. to output the next digit of the remainder.
RESULT	An array used by MULT to accumulate the result.
RLOAD	Loads a screen relative to the current value in BLK
ROLL	(n ---) Rotates the top N stack items.
ROUND	(n1 --- n2) Leaves n1 rounded to nearest 10.
RP@	(--- n) Returns the current value of the return stack pointer.
RVS	(---) An example to reverse the video of a 2K memory-mapped display system.
SEEK	(n1 n2 n3 --- n4 f) Searches for n1 between n2 and n3 and returns index to n1 and true flag if found.
SETUP	(---) Sets up the factorial buffer for FACT.
SGN	(n1 --- n2) Returns the sign of n1, or 0 if zero.
SIEVE	(---) A sieve to filter out numbers with factors used in conjunction with PRIMES.
SIN	(n1 --- n2) Returns sine (scaled by 10000) of angle n1 which, depending on definition used, may be in degrees or radians.
SIZE	1. A constant used by PRIMES.
SIZE	2. A constant used by FACT.
SORT	(n1 n2 ---) Sorts an array of bytes from address n1 to n2.
SQRT	(n1 --- n2) Returns the square root of n1.
SR	Adds a fixed offset for 6522 VIA Shift Register base address to leave the Shift Register address.
SUM	(n ---) Used by MULT to add the partial product. Adds the

	content of *BUFF offset by n to accumulate in RESULT buffer.
T1C-H	Adds a fixed offset for 6522 VIA Timer 1 Counter register High byte address.
T1C-L	As T1C-H but for Low byte.
T1L-H	As T1C-H but for Latch register High byte address.
T1L-L	As T1C-H but for Latch register Low byte address.
T2C-H	As T1C-H but for Timer 2 Counter High byte address.
T2C-L	As T1C-H but for Timer 2 Counter Low byte address.
TABLE	A defining word to create a table of bytes.
THRU	Loads a range of screens.
TIMES	(---) Used by MULT to multiply the contents of 1BUFF and 2BUFF.
TOTHE	An internal definition used by FACTORS to apply trial divisors.
TRIAL	A table of primes numbers used by FACTORS.
TRUE	(--- tf) A constant representing the value of the true flag.
TWIXT	(n lo hi --- f) Returns true flag if n is twixt low and high.
U.	(u ---) Prints the top stack item as an unsigned number.
U.R	(u n ---) as U. but with a field-width of n.
UD.R	(ud n ---) Prints as unsigned the double number ud in a field-width of n.
UM*	(d1 u --- d2) Leaves a double number which is the product of a double-precision and a single-precision number.
UM/	(d1 u --- d2) Leaves a double number which is the quotient of a double-precision and a single-precision number.
UPPER	A variable used by the binary search routine SEEK.
VSWAP	(addr1 addr2 ---) Swaps the address contents.
WEEK1	(n1 --- n2) Formats one week for CALENDAR if n1 is in range of permissible dates.
WITHIN	(lo hi n --- f) tests for N within the limits lo–hi.
WKSPC	(--- addr) Returns the address of a Workspace for use by FP/.
XS	A variable used for storing X-squared in evaluating a series.
[COS]	As for COS but for 0 to ±90 degrees.
[DCOS]	As for COS but for 0 to ±90 degrees in × 65 536 format.
\	Suspend compilation until next line. Pronounced 'skip line'. Generally used to start a comment field.

Bibliography

Anderson, A. and Tracy, M. 1984 *Mastering Forth.* Bowie: Robert J. Bradie Co.

Armstrong, M.A. 1985. *Learning Forth.* John Wiley & Sons

Baker, L. and Derick, M. 1983. *Pocket Guide to Forth.* Reading, Mass: Addison-Wesley

Bishop, O. 1984. *Exploring Forth.* Granada Publishing

Brodie, L. 1981. *Starting Forth.* London: Prentice-Hall International

Brodie, L. 1984. *Thinking Forth.* London: Prentice-Hall International

Chirlian, P.M. 1983. *Beginning Forth.* Dilithium Press

Derick, M. and Baker, L. 1982. *Forth Encyclopedia.* Mountain View Press

Emery, G. 1985. *The Students Forth.* Blackwell Computer Science Texts

Feierback, G. and Thomas, P. 1985. *Forth Tools and Applications*

Haydon, G. 1982. *All about Forth.* Mountain View Press

Huang, T. 1983. *And So Forth.*

Husband, D. 1983. *Advanced Forth.* Wilmslow: Sigma Technical Press

Kail, P.A.C. 1985. *Go Forth — An Introduction to Forth.* Microbooks

Loeliger, R.G. 1981. *Threaded Interpreted Languages.* London: McGraw-Hill

Martin, T. 1985. *Bibliography of Forth References.* 2nd edition

McCabe, K.C. 1983. *Forth Fundamentals.* I. 'Language Usage'. Vol. 2. 'Language Glossary'. Dilithium Press

Olney, R. and Benson, M. 1985. *Forth Techniques.* Pan/Personal Computer News

Olney, R. and Benson, M. 1985. *Fundamental Forth.* Pan/Personal Computer News

Salman, W., Tisserand, O. and Toulout, B. 1985. *Forth.* New York: Springer-Verlag

Scanlon, L.J. 1982. *Forth Programming.* Howard Sams

Scanlon, L.J. 1983. *Forth Programming.* Prentice-Hall International

Ting, C.H. 1981. *Systems Guide to FigForth.* Offete Enterprises

Ting, C.H. 1984. *Forth Notebook.* Offete Enterprises

Ting, C.H. 1985. *Inside Forth-'83.* Offete Enterprises

Vickers, S. 1984. *Pocket Guide: Forth.* London: Pitman Publishing

Winfield, A. 1983. *The Complete Forth.* Wilmslow: Sigma Technical Press

Index

Forth: The NEXT Step

Ron Geere

Forth, the language of the International Astronomical Union, is rapidly establishing itself as an important applications programming language in the professional field, particularly for control applications such as process control and robotics. Apart from its adaptability to a number of diverse applications, Forth has the advantage of being highly portable from one machine to another. In this book, the author concentrates on the two dominant standards of the language, FigForth and '79 - standard, though the recent '83 - standard and other variants are also considered.

Forth, like a living language, is extensible. New 'words' can be defined to suit the particular purpose of the user. In this book the author presents some common extensions to the minimum required word set along with some programs that have been found to be either useful or entertaining. These will stimulate users to discover solutions to their own programming problems. In particular, solutions are provided to problems not readily realized in integer mathematics.

This book will be of interest to the growing body of Forth enthusiasts who, having learned the fundamentals of the language, wish to take the next step - especially scientists, engineers and computing professionals who are looking for 'tools' to do a particular job. The approach throughout is practical, with an emphasis on *using* the language.

Among the useful topics covered are:

★ Formatting

★ Calendar functions

★ Double-number definitions

★ Mathematical functions

Additionally, a wide range of general purpose and frequently used definitions and tools are included. A glossary and bibliography complete this compact and useful reference text.

Addison-Wesley Publishing Company ISBN 0-201-18050-2